NASA Contractor Report 4527

A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model

Patrick Lincoln and John Rushby

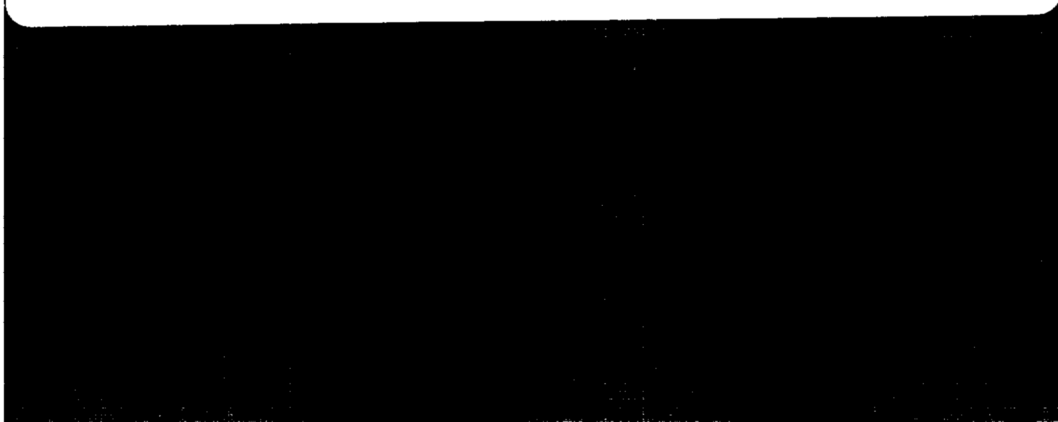
CONTRACT NAS1-18969
JULY 1993

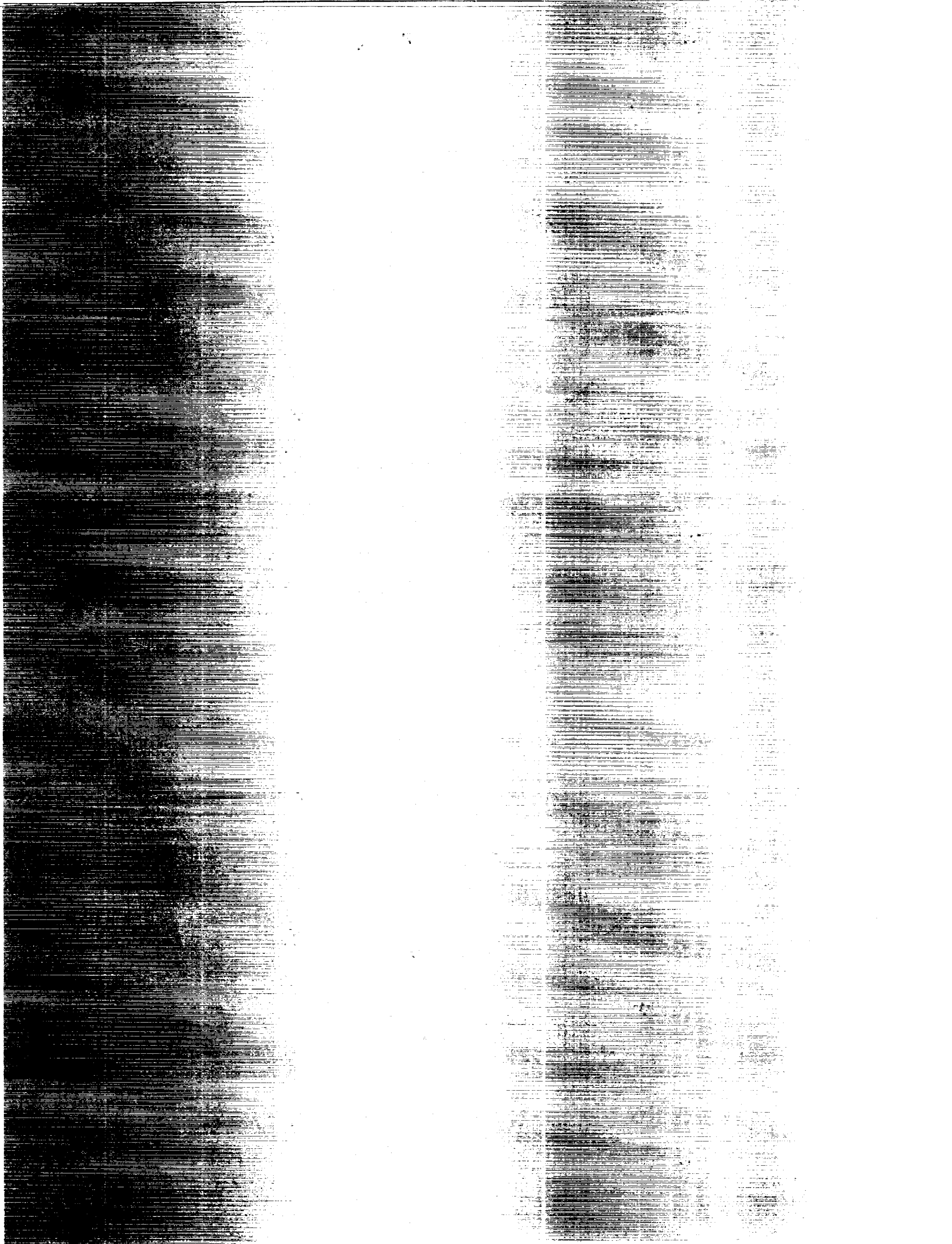
(NASA-CR-4527) A FORMALLY VERIFIED ALGORITHM FOR INTERACTIVE CONSISTENCY UNDER A HYBRID FAULT MODEL Final Report (SRI International Corp.) 72 p

N94-11188

Unclas

H1/61 0179695





NASA Contractor Report 4527

A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model

Patrick Lincoln and John Rushby
SRI International
Menlo Park, California

Prepared for
Langley Research Center
under Contract NAS1-18969

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1993

Abstract

Consistent distribution of single-source data to replicated computing channels is a fundamental problem in fault-tolerant system design. The "Oral Messages" (OM) algorithm of Lamport, Shostak, and Pease solves this problem of Interactive Consistency (also known as Source Congruence or Byzantine Agreement) in the presence of m arbitrary (i.e., Byzantine) Faults, using $m + 1$ rounds of message exchange and $n > 3m$ channels.

A deficiency of OM and similar algorithms is that all faults are assumed to exhibit worst-case (i.e., arbitrary) behavior, so that the algorithm can tolerate no more "simple" faults than truly Byzantine ones. To overcome this deficiency, Thambidurai and Park introduced a "hybrid" fault model that distinguished three fault modes: asymmetric (Byzantine), symmetric, and benign; they also exhibited, along with an informal "proof of correctness," a modified version of OM that withstands a asymmetric, s symmetric, and b benign faults simultaneously, using $m + 1$ rounds, provided $n > 2a + 2s + b + m$, and $m \geq a$.

Unfortunately, this algorithm is flawed; it fails, for example, in the case $n = 5, m = 1$ when the transmitter has a benign fault and one of the receivers is Byzantine. We detected this flaw while undertaking a formal verification of the algorithm using our PVS mechanical verification system. Repairing this algorithm is not easy. We developed an incorrect version ourselves, and even "proved" it correct using ordinary, informal mathematics.

The discipline of mechanically checked formal verification eventually enabled us to develop a correct algorithm for Interactive Consistency under the hybrid fault model. We present this algorithm, discuss its subtle points, and describe its formal specification and verification. Because informal proofs seem unreliable in this domain, and because the consequences of failure could be catastrophic, we believe formal verification should become standard for algorithms intended for safety-critical applications. We argue that formal verification systems such as PVS are now sufficiently effective that their application to such problems may be considered routine.

Keywords: interactive consistency, Byzantine agreement, hybrid fault models, formal verification.

INTENTIONALLY

Contents

1 Introduction **1**

1.1 Related Work 4

2 Requirements, Assumptions, and the Algorithms OM and Z **6**

2.1 Requirements 6

2.2 Assumptions 7

2.3 Algorithm OM 7

2.4 Algorithm Z 8

3 The Algorithm OMH **10**

3.1 Hybrid Fault Model 10

3.2 Repairing Algorithm Z 11

3.3 Semiformal Analysis and Correctness Arguments 13

3.4 Extreme Cases 16

3.5 Benefits 17

3.6 Communications Faults 18

3.7 Implementing R and UnR 20

4 The Formal Specification and Verification **21**

4.1 Formal Specification 21

4.2 Formal Verification 35

4.2.1 Portion of PVS Proof of Validity 38

4.3 PVS Proof Chain Analysis 47

5 Conclusions **53**

A Hybrid MJRTY **60**

A.1 Using Hybrid_mjrty 66

IV INTENTIONALLY

Chapter 1

Introduction

Fault-tolerant systems are designed and evaluated against explicit assumptions regarding the kinds and numbers of faults they are to tolerate. "Fault models," which enumerate the assumed behaviors of faulty components, range from those that identify many highly specific modes of failure, to those that comprise just a few broad classes. The advantage of a very detailed fault model is that the mechanisms of fault tolerance can be finely tuned to deliver maximum resilience from a given level of redundancy; the corresponding disadvantages are that an overlooked fault mode may cause unexpected failure in operation, and the need to counter many fault modes can lead to a complex design—which may itself be a source of faults.

In contrast to designs that consider many fault modes are those that make no assumptions whatsoever about the behavior of faulty components. The advantage of such "Byzantine" fault-tolerant designs is that they cannot be defeated by unexpected failure modes; their disadvantage is that all faults are treated as "worst case," so that large levels of redundancy tolerate relatively few faults. For example, a conventional Byzantine fault-tolerant architecture requires $3m+1$ channels to tolerate m simultaneous faults of any kind within some of its functions [25,1]. Thus, a 4-plex is needed in order to withstand a single fault,[1] and 5- and 6-plexes provide no additional benefit (in fact the additional channels will increase the fault arrival rate and thereby lower overall reliability).[2] This seems counterintuitive, since it is clear that suitably organized 5- and 6-plexes can withstand more faults, of some kinds, than a 4-plex.

These observations motivate the study of fault-tolerant architectures and algorithms with respect to hybrid fault models that include the Byzantine, or "arbitrary," fault mode, together with a limited number of additional fault modes.

[1] Single fault tolerance can also be provided by architectures such as Draper Laboratory's FTP, which uses only three full processors plus three simpler "interstages" [13].

[2] If the system can be reconfigured following a fault, then a fifth and sixth channel can increase reliability if used as standby spares—but they serve no purpose as live channels.

Inclusion of the arbitrary fault mode (i.e., faults whose behaviors are entirely unconstrained) eliminates the fear that some unforeseen mode may defeat the fault-tolerance mechanisms provided, while inclusion of other fault modes allows greater resilience to be achieved for faults of these kinds than with a classical Byzantine fault-tolerant architecture.

Our interest is architectures for digital flight-control systems, where fault-masking behavior is required to ultra-high levels of reliability. This means that not only must stochastic modeling show that adequate numbers and kinds of faults are masked to satisfy the mission requirements, but that convincing analytical evidence must attest to the soundness of the overall fault-tolerant architecture and to the correctness of the design and implementation of its mechanisms of fault tolerance.[3] Such a rational design for a "reliable computing platform" suitable for ultra-critical applications was established in the late 1970s and early 1980s by the SIFT architecture [36]: the system workload is executed by several independent channels operating in approximate synchrony, and results are subjected to majority voting. If all channels execute identical workloads on identical data, then majority voting is sufficient to mask arbitrary channel failures. However, majority voting is not sufficient to mask arbitrary failures in the distribution of single-source data (such as sensor samples) [25], nor in clock synchronization [16].

In this report, we focus on algorithms for reliably distributing single-source data to multiple channels in the presence of faults. This problem, known as "Interactive Consistency" (although sometimes called "source congruence"), was first posed and solved for the case where faulty channels can exhibit arbitrary behavior by Pease, Shostak, and Lamport [25] in 1980.[4] Interactive Consistency is a symmetric problem: it is assumed that each channel has a "private value" (e.g., a set of sensor samples) and the goal is to ensure that every nonfaulty channel achieves an accurate record of the private value of every other nonfaulty channel. In 1982, Lamport, Shostak, and Pease [17] presented an asymmetric version of Interactive Consistency, which they called the "Byzantine Generals Problem"; here, the goal is to communicate a single value from a designated channel called the "Commanding General" to all the other channels, which are known as "Lieutenant Generals." The problem of real practical interest is Interactive Consistency, but the metaphor of the Byzantine Generals has proved so memorable that this formulation is better known; it can also be easier to describe algorithms informally using the Byzantine Generals formulation, although the balance of advantage can be reversed in truly formal presentations [27]. An algorithm for the Byzantine Generals problem can be converted to one for Interactive Consistency by simply iterating it over all channels

[3] There are examples where unanticipated behaviors of the mechanisms for fault tolerance became the *primary* source of system failure [18].

[4] Davies and Wakerley had anticipated some of the issues a few years earlier [9].

(each channel in turn taking the role of the Commander), so there is no disadvantage to considering the Byzantine Generals formulation. See [27] for more extended discussion of this topic.

Lamport, Pease, and Shostak presented algorithms for solving the Byzantine Generals problem. The principal difficulty to be overcome in such algorithms is possibly asymmetric behavior on the part of faulty channels: such a channel may provide one value to a second channel, but a different value to a third, thereby making it difficult for the recipients to agree on a common value. Byzantine Generals algorithms overcome the possibility of faulty channels exhibiting asymmetric behavior by using several rounds of message exchange during which channel p tells channel q what value it received from channel r and so on. The precise form of the algorithm depends on assumptions about what a faulty channel may do when relaying such a message; under the "Oral Messages" assumption, there is no guarantee that a faulty channel will relay messages correctly. This corresponds to totally arbitrary behavior by faulty channels: not only can a faulty channel provide inconsistent data initially, but it can also relay data inconsistently.[5]

Using $m+1$ rounds of message exchanges, the Oral Messages algorithm of Lamport, Shostak, and Pease [17], which we denote OM(m), can withstand up to m arbitrary faults, provided n, the number of channels, satisfies $n > 3m$. The bound $n > 3m$ is optimal: Pease, Shostak, and Lamport proved that no algorithm based on the Oral Messages assumptions can withstand more arbitrary faults than this [25]. However, as we have already noted, OM(m) is not optimal when other than arbitrary faults are considered: other algorithms can withstand greater numbers of simpler faults for a given number of channels than OM(m).

We are not the first to make these observations. Thambidurai and Park [33] and Meyer and Pradhan [20,21] have considered Interactive Consistency algorithms that resist multiple fault classes. Thambidurai and Park's "Unified" model divides faults into three classes: *nonmalicious* (or *benign*), *symmetric malicious*, and *asymmetric malicious*. A nonmalicious fault is one that produces detectably missing values (e.g., timing, omission, or crash faults), or that produces a "self-incriminating" value that all nonfaulty recipients can detect as bad (e.g., it fails checksum or format tests). A malicious fault is one that yields a value that is not detectably bad (i.e., it is a *wrong*, rather than a missing or manifestly corrupted, value). A symmetric malicious fault delivers the same wrong value to every nonfaulty receiver; an asymmetric malicious fault delivers (possibly) different wrong values (or missing or detectably bad values) to different nonfaulty receivers. The classical arbitrary or Byzantine fault is an asymmetric malicious fault in this classification. Note that a nonmalicious fault may be asymmetric in that different detectably bad values may be sent, but if a

[5] Under the "signed messages" assumption (which can be satisfied using digital signatures), an altered message can be detected by the recipient.

good value is sent to some receivers and manifestly bad values are sent to all others, then the sender is considered asymmetric malicious in this taxonomy.

Thambidurai and Park present a variant on the classical Oral Messages algorithm that retains the effectiveness of that algorithm with respect to arbitrary faults, but that is also capable of withstanding more faults of the other kinds considered.[6] In a later paper, Thambidurai, Park, and Trivedi [34] present reliability analyses that show this increased fault tolerance indeed provides superior reliability under plausible assumptions. McElvany-Hugue has also studied the reliability of related algorithms under this fault model, reaching similar conclusions [14].

Unfortunately, Thambidurai and Park's algorithm (which they call Algorithm Z) has a serious flaw and fails in quite simple circumstances. In this report, we describe the flaw, and explain how straightforward attempts to repair it also fail. We then present a correct algorithm for the problem of Interactive Consistency under a hybrid fault model and present a proof of its correctness. Thambidurai and Park presented a proof of correctness for their flawed algorithm, and we have also developed some rather convincing "proofs" ourselves for other incorrect algorithms for this problem. Accordingly, we have developed a mechanically checked formal verification for our algorithm using the PVS verification system [22]. (In fact, all this work grew out of the attempt to formally verify their Algorithm Z.) We describe this formal verification and show that it is not particularly difficult. Because informal proofs seem unreliable in this domain, and the consequences of failure could be catastrophic, we argue that formal verification should become routine.

1.1 Related Work

Hybrid fault models were first introduced for the MAFT architecture, developed at Allied-Signal's Aerospace Technology Center [15]. MAFT provides a rich set of fault-tolerant mechanisms and services, including clock synchronization (both steady-state and startup), interactive consistency with both exact and approximate agreement, and sophisticated error detection and reconfiguration [35]. The interactive-consistency algorithm employed in MAFT apparently does not suffer from the problem we identified in Algorithm Z, and Michelle McElvany Hugue and others at Allied Signal have developed corrected versions of Algorithm Z and proofs of correctness that are similar to ours.

The reason we selected hybrid fault models and their algorithms for study is that these seem among the most interesting and useful directions in current research for

[6] Meyer and Pradhan [21] consider a fault model that, in Thambidurai and Park's taxonomy, comprises only asymmetric malicious and benign faults. Their algorithm is derived from the algorithm of [11] and, like the parent algorithm, is not particularly suitable for the cases of practical interest (i.e., $m = 1$, or possibly $m = 2$, n less than 10).

fault-tolerant architectures. However, our interest is less in the algorithms themselves, and more in the use of mechanically-checked formal methods as a systematic and rigorous means to analyze these algorithms, to identify all the assumptions on which they depend, to detect and help correct errors in their formulation, and to provide compelling arguments for their correctness. From this point of view, algorithms for hybrid fault models are particularly interesting because of their subtlety, and the extended case analysis required in their analysis. As with other algorithms where we have discovered errors in published analyses [28], our corrections are technical adjustments (though quite radical in the case of the algorithm considered here) that build on the insights, algorithms, and analyses of the original authors.

Chapter 2

Requirements, Assumptions, and the Algorithms OM and Z

Although the problem of real practical interest is Interactive Consistency, all the algorithms we consider are presented here in their Byzantine Generals formulation, since this appears simpler in informal presentations. The relationship between Interactive Consistency and the Byzantine Generals Problem is examined in [27].

2.1 Requirements

In the Byzantine Generals formulation of the problem, there are n participants, which we call "processors." A distinguished processor, which we call the *transmitter*, possesses a value to be communicated to all the other processors, which we call the *receivers*.[1] There are n processors in total, of which some (possibly including the transmitter) may be faulty. The transmitter's value is denoted v and the problem is to devise an algorithm that will allow each receiver p to compute an estimate ν_p of the transmitter's value satisfying the following conditions:

BG1: If receivers p and q are nonfaulty, then they agree on the value ascribed to the transmitter—that is, for all nonfaulty p and q, $\nu_p = \nu_q$.

BG2: If the transmitter is nonfaulty, then every nonfaulty receiver computes the correct value—that is, for all nonfaulty p, $\nu_p = v$.

Conditions BG1 and BG2 are sometimes known as "Agreement" and "Validity," respectively.

[1]Lamport, Shostak, and Pease [17] speak of a "Commanding General" and of "Lieutenant Generals" where we say transmitter and receivers.

2.2 Assumptions

The principal difficulty that must be overcome by a Byzantine Generals algorithm is that the transmitter may send different values to different receivers, thereby complicating satisfaction of condition BG1. To overcome this, algorithms use several "rounds" of message exchange during which processor p tells processor q what value it received from processor r and so on. Under the "Oral Messages" assumptions, the difficulty is compounded because a faulty processor q may "lie" to processor r about the value it received from processor p. More precisely, the *Oral Messages* assumptions are the following.

A1: Every message that is sent between nonfaulty processors is correctly delivered.

A2: The receiver of a message knows who sent it.

A3: The absence of a message can be detected.

In the classical Byzantine Generals problem, there are no constraints at all on the behavior of a faulty processor.

2.3 Algorithm OM

Lamport, Shostak, and Pease's Algorithm OM solves the Byzantine Generals problem under the Oral Messages assumption. The algorithm is parameterized by m, the number of rounds of message exchanges performed. OM(m) can withstand up to m faults, provided $n > 3m$, where n is the total number of processors. The algorithm is described recursively; the base case is OM(0).

OM(0)

1. The transmitter sends its value to every receiver.
2. Each receiver uses the value obtained from the transmitter, or some arbitrary, but fixed, value if nothing is received.

Now we can describe the general case.

OM(m), m > 0

1. The transmitter sends its value to every receiver.
2. For each p, let v_p be the value receiver p obtains from the transmitter, or else be some arbitrary, but fixed, value if it obtains no value. Each receiver p acts as the transmitter in Algorithm OM($m-1$) to communicate its value v_p to each of the $n-2$ other receivers.

3. For each p, and each $q \neq p$, let v_q be the value receiver p obtained from receiver q in step (2) (using Algorithm OM($m-1$)), or else some arbitrary, but fixed, value if nothing was received. Each receiver p calculates the majority value among all values v_q it receives, and uses that as the transmitter's value (or some arbitrary, but fixed, value if no absolute majority exists).

The correctness of this algorithm (that it achieves BG1 and BG2 under certain assumptions) was proven in [17, page 390] and mechanically checked in [1,27].

2.4 Algorithm Z

Thambidurai and Park's Algorithm Z is a modification of OM intended to operate under their hybrid fault model described earlier. The difference between OM and Z is that the latter has a distinguished "error" value, E. Any processor that receives a missing or manifestly bad value replaces that value by E and uses E as the value that it passes on in the recursive instances of the algorithm. The majority voting that is required in OM, is replaced in Z by a majority vote with all E values eliminated. Thambidurai and Park claim that an m-round implementation of Algorithm Z can withstand $a + s + b$ simultaneous faults, where a is the number of asymmetric malicious faults, s the number of symmetric malicious faults, and b the number of nonmalicious faults, provided $a \leq m$, and n, the number of processors, satisfies $n > 2a + 2s + b + m$. In the case of no symmetric malicious or nonmalicious faults (i.e., Byzantine faults only), we have $m = a$ and $s = b = 0$, so that $n > 3m$ and the algorithm provides the same performance as the classical Oral Messages algorithm.

We and our colleagues at SRI have undertaken mechanically checked formal verifications for a number of fault-tolerant algorithms, including OM [27], and have identified deficiencies in some of the previously published analyses (though not in the algorithms—see, for example [24,29,30]). Any changes to the established algorithms for Interactive Consistency must be subjected to intense scrutiny, for errors in these algorithms are single points of failure in any system that employs them. Changes that widen the classification of faults considered are likely to increase the case analysis, and hence the complexity and potential fallibility of arguments for the correctness of modified algorithms. We therefore considered Thambidurai and Park's Algorithm Z an interesting candidate for formal verification.

We began our attempt to formally verify Algorithm Z by studying the proof of its correctness provided by Thambidurai and Park [33, pages 96 and 97]. This proof follows the outline of the standard proof for OM [17, page 390] quite closely. However, we soon found that Thambidurai and Park's proof is simultaneously more complicated than necessary and flawed in several details. The most serious fault

is that their Lemma 1 (all nonfaulty receivers get the correct value of a nonfaulty transmitter) fails to consider the case where the value sent by the transmitter is E. This can arise in recursive instances of the algorithm when nonfaulty receivers are passing on the value received from a faulty source. Further thought soon reveals that not only is the proof flawed, but the algorithm is incorrect: even systems with large numbers of processors may fail with only two faulty components.

The simplest counterexample comprises five processors in which the transmitter has a nonmalicious fault, one of the receivers has an asymmetric malicious fault, and the algorithm is Z with one round (i.e., $n = 5, a = 1, s = 0, b = 1, m = 1$). All the nonfaulty receivers note E as the value received from the transmitter, and relay the value E to all the other receivers. The faulty receiver sends a different (non-E) value to each of the nonfaulty receivers. Each nonfaulty receiver then has three E values, and one non-E value; because E values are discarded in the majority vote, each nonfaulty receiver selects the value received from the faulty receiver as the value sent by the transmitter. Since these values are all different, the algorithm has failed to achieve agreement among the nonfaulty receivers. Observe that this scenario is independent of the number of receivers (provided there are more than three of them—two that should agree and one that is faulty), so the problem is not due to an inadequate level of redundancy.

Chapter 3

The Algorithm OMH

OMH is our new algorithm for Interactive Consistency under a hybrid fault model. In this chapter, we present the fault model, the algorithm, and its correctness properties semiformally; the mechanically-checked formal specification and verification are described in the next chapter.

3.1 Hybrid Fault Model

Our fault model is that of Thambidurai and Park, but with the cases renamed—we find the anthropomorphism in terms such as "malicious faults" unhelpful.

The fault modes we distinguish for processors are *arbitrary-faulty*, *symmetric-faulty*, and *manifest-faulty* (also called *crash-faulty*). (These correspond to Thambidurai and Park's asymmetric malicious, symmetric malicious, and nonmalicious faults, respectively.) Of course, we also need a class of *good* (also called *nonfaulty*) processors. We specify these fault modes semiformally as follows (the formal characterizations are presented in the following chapter).

When a transmitter sends its value v to the receivers, the value obtained by a nonfaulty receiver p is:

- v, if the transmitter is nonfaulty
- E, if the transmitter is manifest-faulty[1]
- Unknown, if the transmitter is symmetric-faulty, but all receivers obtain the *same* value,

[1]Some preprocessing of timeouts, parity and "reasonableness" checks, etc. may be necessary to identify manifestly faulty values. The intended interpretation is that the receiver detects the incoming value as missing or bad, and then replaces it by the distinguished value E.

- Completely unconstrained, if the transmitter is arbitrary-faulty.

Note that it is not necessary to define the value received by a faulty receiver, because such receivers may send values completely unrelated to their inputs. Also note that manifest faults must be symmetric. If a processor were to "crash" during this protocol (or if some of its outgoing links are broken, or if it is early or late transmitting on some links), it would have to be counted as arbitrary-faulty, since different good receivers may obtain different values—even though the values sent are either correct or identifiably bad. It is possible to treat such cases as a new class of faults, which, depending on practical considerations, may be an interesting area for future research (see Section 3.6).

3.2 Repairing Algorithm Z

It seems that the flaw in Algorithm Z stems from the fact that it does not distinguish between values received from manifest-faulty processors and the *report* of such values received from nonfaulty processors; the single value E is used for both cases. Thus, a plausible repair for Algorithm Z introduces an additional distinguished value RE (for Reported Error); when a manifestly faulty value is received, the receiver notes it as E, but passes it on as RE; if an RE is received, it is noted and passed on as such. Only E values are discarded when the majority vote is taken. In the counterexample to Algorithm Z given above, the nonfaulty receivers in this modified algorithm will each interpret the value received from the transmitter as E, and pass it on to the other receivers as RE. In their majority votes, each nonfaulty receiver has a single E (from the transmitter), which it discards, two REs (from the other nonfaulty receivers), and an arbitrary value (from the faulty receiver). All will therefore select RE as the value ascribed to the transmitter.

Unfortunately this modified algorithm has two defects. First, a receiver that obtains a manifest-faulty value from the transmitter notes it as E, but passes it on as RE. Thus, this receiver will omit the value from its majority vote, but the others will include it (as RE). This asymmetry can be exploited by an arbitrary-faulty transmitter to force the receivers into disagreement (consider an arbitrary-faulty transmitter and three nonfaulty receivers, where the transmitter sends the values E, RE, and a normal value).

It therefore seems that receivers must distinguish between an E received from the transmitter (which must be treated locally as RE and passed on as such), and one received from another receiver (which can be discarded in the majority vote). This repair fixes one problem, but leaves the other: the value ascribed to a manifest faulty transmitter is not E, but RE. This might seem a small inconvenience, but it causes the algorithm to fail when m, the number of rounds, is greater than 1

(consider the case $n = 6$, $m = 2$ when there is a nonfaulty transmitter and three manifest-faulty receivers).

A repair to this difficulty might be to return the value E whenever the majority vote yields the value RE. This modification has the problem that receivers cannot distinguish a manifest-faulty receiver from a nonfaulty one reporting that another is manifest-faulty (consider the case $n = 4$, $m = 1$, all the processors are nonfaulty, and the transmitter is trying to send RE—as can arise in recursive cases when $m > 1$).

Like Thambidurai and Park did for Algorithm Z, we produced rather convincing, but nonetheless flawed, informal "proofs of correctness" for these erroneous repairs to Algorithm Z. Eventually, the discipline of formal verification (where one must deal with the implacable skepticism of a mechanical proof checker and is eventually forced to confront overlooked cases and unstated assumptions) enabled us to develop a genuinely correct algorithm for this problem.

Our new algorithm, OMH (for "Oral Messages, Hybrid"), is somewhat related to the last of the modifications to Algorithm Z indicated above, but recognizes that a single "reported error" value is insufficient. OMH employs two functions R and UnR that act as a "wrapper" and an "unwrapper" for error values.

The basic idea of OMH is that at each round, the processors do not forward the actual value they received. Instead, each processor sends a value corresponding to the statement "I'm reporting *value*." One can imagine that after several rounds, messages corresponding to "I'm reporting that he's reporting that she's reporting an Error value" arise. This wrapper is only required for error values, but for simplicity we assume that the functions R and UnR are applied to *all* values. Alternatives to this are explored in Section 3.7. This leaves the following intuitive picture of the algorithm.

Proceed as in the usual OM Byzantine agreement algorithm presented above, with the following exceptions. Add a distinguished error value E, and two functions on values R and UnR. When a manifestly bad value is received, temporarily record it as the special value E.

When passing along a value received from the transmitter or incorporating it into the local majority vote, apply R, standing for "I report..." to the value. Discard all E values (received from other receivers) before voting, but treat all other error values ($R(E)$, $R(R(E))$, etc.) as normal, potentially valid values during voting. After voting, apply UnR (strip off one R) before returning the value.

The key idea here is that in Z and related algorithms there is a confusion about which processors have manifest faults: if there is only one error value, E, how can a processor distinguish between a manifest-faulty receiver and a good receiver reporting a bad value (or the lack of a value) from a manifest-faulty transmitter? The counterexample to Algorithm Z given above exploits this confusion, but it is handled

correctly by OMH, because the nonfaulty receivers in OMH(1) each receive a single E from the transmitter, which they pass on to the other receivers and themselves as $R(E)$. The values thus voted on include three $R(E)$s and an arbitrary value (from the arbitrary-faulty receiver). All nonfaulty receivers therefore select $R(E)$ as the majority value. After stripping one R from this value, the result correctly identifies the transmitter as manifest-faulty. In short, OMH incorporates the diagnosis of manifest faults into the agreement algorithm.

The Hybrid Oral Messages Algorithm OMH(m) is defined more formally below, and completely formally in Chapter 4.1:

OMH(0)

1. The transmitter sends its value to every receiver.
2. Each receiver uses the value received from the transmitter, or uses the value E if a missing or manifestly erroneous value is received.

OMH(m), m > 0

1. The transmitter sends its value to every receiver.
2. For each p, let v_p be the value receiver p obtains from the transmitter, or E if no value, or a manifestly bad value, is received.

   Each receiver p acts as the transmitter in Algorithm OMH($m-1$) to communicate the value $R(v_p)$ to all of the $n-1$ receivers, including itself.
3. For each p and q, let v_q be the value receiver p received from receiver q in step (2) (using Algorithm OMH($m-1$)), or else E if no such value, or a manifestly bad value, was received. Each receiver p calculates the majority value among all non-error values v_q received, (if no majority exists, the receiver uses some arbitrary, but functionally determined value) and then applies UnR to that value, using the result as the transmitter's value.

3.3 Semiformal Analysis and Correctness Arguments

We make explicit a few unsurprising technical assumptions:

- All processors are either nonfaulty, arbitrary-faulty, symmetric-faulty, or manifest-faulty. (Any fault not otherwise classified is considered arbitrary.)

- Processors do not change fault status during the procedure; for example, if a nonfaulty processor were to become manifest-faulty during this procedure, we would say that processor is arbitrary-faulty because it has effectively sent different values to other processors.
- For all values v, $R(v) \neq E$. (Wrapped values are never mistaken for errors.)
- For all values v, $UnR(R(v)) = v$. (Unwrapping a wrapped value results in the original value.)

Algorithm OMH must satisfy the Byzantine Generals conditions naturally extended to the fault model described above.

When the transmitter is symmetric-faulty, it is convenient to call the unique value received by all nonfaulty receivers the value *actually sent* by the transmitter.

BGH1: If processors p and q are nonfaulty, then they agree on the value ascribed to the transmitter; that is, $\nu_p = \nu_q$.

BGH2: If processor p is nonfaulty, the value ascribed to the transmitter by p is

- The correct value v, if the transmitter is nonfaulty,
- The value actually sent, if the transmitter is symmetric-faulty,
- The value E, if the transmitter is manifest-faulty.

The argument for the correctness of OMH is an adaptation of that for the Byzantine Generals formulation of OM [17, page 390]. We define

- n, the number of processors,
- a, the maximum number of arbitrary-faulty processors the algorithm is to tolerate,
- s, the maximum number of symmetric-faulty processors the algorithm is to tolerate,
- c, the maximum number of manifest-faulty processors the algorithm is to tolerate,[2]
- m, the number of rounds the algorithm is to perform.

[2]We cannot use m for the number of manifest-faulty processors, because the parameter m is traditionally used for the number of rounds (although Thambidurai and Park use r). The symbol c can be considered a mnemonic for "crashed," which is one of the failures that can generate manifest-faulty behavior.

Lemma 1 *For any a, s, c and m, Algorithm OMH(m) satisfies BGH2 if there are more than $2(a+s)+c+m$ processors.*

Proof: The proof is by induction on m. BGH2 specifies only what must happen if the transmitter is not arbitrary-faulty. In the base case $m = 0$, a nonfaulty receiver obtains the transmitter's value if the transmitter is nonfaulty. If the transmitter is symmetric-faulty the value obtained is the value actually sent. If the transmitter is manifest-faulty the receiver obtains the value E. So the trivial algorithm OMH(0) works as advertised and the lemma is true for $m = 0$. We now assume the lemma is true for $m-1$ $(m > 0)$, and prove it for m.

In step (1) of the algorithm, the transmitter effectively sends some value ν to all $n-1$ receivers. If the transmitter is nonfaulty, ν will be v, the correct value; if it is symmetric-faulty, ν is the value actually sent; if it is manifest-faulty, ν is E. In any case, we want all the nonfaulty receivers to decide on ν.

In step (2), each receiver applies OMH($m-1$) with $n-1$ participants. Those receivers that are nonfaulty will apply the algorithm to the value $R(\nu)$. Since by hypothesis $n > 2(a+s)+c+m$, we have $n-1 > 2(a+s)+c+(m-1)$, so we can apply the induction hypothesis to conclude that the nonfaulty receiver p gets $v_q = R(\nu)$ for each nonfaulty receiver q. Let c' denote the number of manifest-faulty processors among the receivers. At most $(a+s+c')$ of the $n-1$ receivers are faulty, so each nonfaulty receiver p obtains a minimum of $n-1-(a+s+c')$ values equal to $R(\nu)$. Since there are c' manifest-faulty processors among the receivers, a nonfaulty receiver p also obtains a minimum of c' values equal to E and, therefore, at most $n-1-c'$ values different from E. The value $R(\nu)$ will therefore win the *hybrid-majority* vote performed by each nonfaulty processor p, provided

$$2(n-1-(a+s+c')) > n-1-c',$$

that is, provided

$$n > 2(a+s)+c'+1.$$

Now, $c' \leq c$, and $1 \leq m$, so this condition is ensured by the constraint

$$n > 2(a+s)+c+m.$$

Finally, *UnR* is applied to the result $R(\nu)$, which results in final value ν. □

Theorem 1 *For any m, Algorithm OMH(m) satisfies conditions BGH1 and BGH2 if there are more than $2(a+s)+c+m$ processors and $m \geq a$.*

Proof: The proof is by induction on m. In the base case $m = 0$ there can be no arbitrary-faulty processors, since $m \geq a$. If there are no arbitrary-faulty processors then the previous lemma ensures that OMH(0) satisfies BGH1 and BGH2. We therefore assume that the theorem is true for OMH($m-1$) and prove it for OMH(m), $m > 0$.

We next consider the case in which the transmitter is not arbitrary-faulty. Then BGH2 is ensured by Lemma 1, and BGH1 follows from BGH2.

Now consider the case where the transmitter is arbitrary-faulty. There are at most a arbitrary-faulty processors, and the transmitter is one of them, so at most $a-1$ of the receivers are arbitrary-faulty. Since there are more than $2(a+s)+c+m$ processors, there are more than $2(a+s)+c+m-1$ receivers, and

$$2(a+s)+c+m-1 > 2([a-1]+s)+c+[m-1].$$

We may therefore apply the induction hypothesis to conclude that OMH($m-1$) satisfies conditions BGH1 and BGH2. Hence, for each q, any two nonfaulty receivers get the same value for v_q in step (3). (This follows from BGH2 if one of the two receivers is processor q, and from BGH1 otherwise). Hence, any two nonfaulty receivers get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value *hybrid-majority*$(v_1, \ldots, v_{n-1})$ in step (3) (since this value is functionally determined), thereby proving BGH1. □

3.4 Extreme Cases

Although a major improvement on OM, the number of faults that can be tolerated by OMH according to the analysis given above is not optimal in some of the extreme circumstances. In some cases, the algorithm is suboptimal; in others, the general analysis given above is too conservative. As an example of the latter, consider the case where only manifest faults are present. In this case, the general analysis above indicates that the number of manifest faults that can be tolerated is $n - m - 1$: in other words, the greater the number of rounds, the *fewer* manifest faults that can be tolerated. In fact, alternative analysis shows that OMH(m) tolerates the maximum possible number of manifest-faulty processors when there are no arbitrary nor symmetric faults. The only constraint is that there must be more processors (whether faulty or not) than rounds (since otherwise some recursive instances would be run on the empty set of processors).

Theorem 2 *If arbitrary and symmetric faults are not present, Algorithm OMH(m) satisfies conditions BGH1 and BGH2 provided there are more than m processors.*[3]

This theorem has been formalized and mechanically verified. The formal proof follows that of Theorem 1 closely, using analogous lemmas. However, here there are only two cases to consider (good and manifest) whereas there are four in the previous argument (good, manifest, symmetric, and arbitrary).

When only symmetric faults are present, it is the algorithm, rather than its general analysis, that is less than optimal. Here, the additional rounds of message exchanges are actively counterproductive in the cases $m > 0$ (compare $n = 4$, $s = 2$ for the cases $m = 0$ and $m = 1$). Additional rounds of messages are the price paid for overcoming arbitrary faults, and these seem to reduce the ability to deal with symmetric faults. An interesting topic for future research is to investigate whether this trade-off can be mitigated.

When no manifest faults are present, Algorithm OMH becomes similar to the traditional Algorithm OM. A related point was made in [33]: in the absence of error values, hybrid majority is equivalent to majority. Thus the only substantive difference between OMH and OM are the wrapper and unwrapper functions applied to values. As discussed in Section 3.7 these functions may be identity on nonerror values, in which case OMH becomes exactly OM in the absence of manifest errors. Thus the analysis above may be applied, showing that the traditional algorithm OM(m) can withstand more faults than is suggested by its standard analysis: in fact, OM(m) satisfies conditions BGH1 and BGH2 if there are more than $2(a+s)+m$ processors, $m \geq a$, and manifest faults are counted as symmetric.

3.5 Benefits

Recall that OM achieves agreement and validity if there are more than three times as many good processors as arbitrary-faulty processors ($n > 3a$). From the bounds given in Theorem 1, $n > 2(a + s) + c + m$ and $m \geq a$, it may be seen that OMH achieves the same resilience to arbitrary faults if there are no symmetric-faulty or manifest-faulty processors. Also, from Theorem 2, if $a = s = 0$, then all that is required is that $n > r$.

While providing the same resilience to arbitrary or Byzantine faults, OMH achieves a higher degree of tolerance to other classes of faults than OM. Table 3.1 indicates the different numbers of simultaneous faults that a 6-plex can withstand using OMH(1); for comparison, observe that OM(1) can withstand only a single

[3] Of course, the conditions are somewhat vacuous if there are less than two good processors.

Number of Faults		
Arbitrary (a)	Symmetric (s)	Manifest (c)
1	1	0
1	0	2
0	2	0
0	1	2
0	0	5

Table 3.1: Fault-Masking Capabilities of OMH(1) with $n = 6$

(arbitrary) fault in this configuration.[4] Thambidurai, Park, and Trivedi [34] present reliability analyses that show this increased fault tolerance indeed provides superior reliability under plausible assumptions[5]. McElvany-Hugue has also studied the reliability of related algorithms under this fault model, reaching similar conclusions [14]. In fact, our crash-only analysis above shows that OMH exhibits slightly greater fault tolerance than that assumed in these reliability analyses.

3.6 Communications Faults

A disadvantage of most fault models for Interactive Consistency, including the one used here, is that they attribute communications failure on a link connecting two processors to one or other of the processors concerned.[6] In the draconian fault-model of the original OM algorithm (i.e., all processors faults are Byzantine) this causes a communications fault on a link—a physical fault that may be considered fairly likely, and relatively innocuous in its effects—to be counted as one of the most difficult, and hopefully rare, of all faults.

It is worth inquiring whether the hybrid model used here can be extended to treat communications faults as other than arbitrary processor faults. The problem in developing a hybrid fault model that includes communications faults as well as arbitrary processor faults, is that a communications fault *does* have the asymmetric character of a Byzantine or arbitrary fault. If we introduce communications faults as a new fault class, different from arbitrary processor faults, then we must account

[4]That is according to the classical analysis. As noted in the previous section, revised analysis of OM(1) shows that it can actually withstand two simultaneous faults, provided at most one of them is arbitrary. The chief difference between OM and OMH is that OM does not distinguish manifest faults from (other) symmetric ones.

[5]Although Algorithm Z is flawed, the analysis in [34] can be correctly applied to OMH

[6]Perry and Toueg [26] presented an interactive-consensus algorithm for a fault model that admits communications failures, but that model does not consider Byzantine faults at all.

for these faults either in the result corresponding to Lemma 1 above, or in that corresponding to Theorem 1. The latter seems the most likely candidate, since it is the case that deals with asymmetrically faulty transmitters. However, the inductive proof used in Theorem 1 does not work for the case of a transmitter with an asymmetric but nonarbitrary communications fault (because in the recursive subcases we will still have the same number of arbitrary faults to mask, but one less round to do it in—and we cannot mask more arbitrary faults than rounds). The alternative is to consider communications faults in Lemma 1. It turns out that this is feasible, but is equivalent to regarding a communications fault as a symmetric-fault in the receiver.

This conclusion seems fairly useful, so we record it in the following definition and theorem.

- Let p and q be processors; if there is a *communications fault* on the *link* $p \rightarrow q$, then a receiver can receive any value (i.e., we allow intermittent and Byzantine behavior).[7]

Theorem 3 *Let C be the set of links with communications faults. Then Theorem 1 can be applied provided that for each $p \rightarrow q \in C$, either:*

- *processor p is counted as arbitrary-faulty (whether it actually is or not), or*
- *processor q is manifest-faulty, symmetric-faulty, or arbitrary-faulty and is counted as such, or*
- *processor q is nonfaulty but is counted as symmetric-faulty.*

Proof: First, we consider the cases where processor q is faulty. In all cases (arbitrary-faulty, symmetric-faulty, and manifest-faulty), the behavior of a faulty processor is independent of the values it receives; hence the faulty link $p \rightarrow q$ is irrelevant.

If q is nonfaulty, we can attribute the faulty link $p \rightarrow q$ to either p or q. If we attribute it to p, then p appears to manifest arbitrary (i.e., Byzantine) behavior, and must be counted as arbitrary-faulty.

If we attribute the link fault $p \rightarrow q$ to q and q is nonfaulty, then the behavior seen by other processors is precisely that of a symmetric-faulty processor: if the link delivers a wrong value (or the correct one) to q, it will faithfully pass it on to all the other receivers; if the link delivers a corrupted (or no) value to q, it will pass on R(E). Thus q should be counted as a symmetric-faulty processor. □

This and related alternative models of link faults, and other simple but asymmetric classes of communication faults, are interesting avenues for further work.

[7] In the treatment used here, there is no advantage in a more restrictive model of communications faults.

3.7 Implementing R and UnR

Although the informal and formal specifications suggests that R and UnR are applied to all values at every round, this is unnecessary. R and UnR may be identity on nonerror values. That is, the following definitions suffice:

$$R(x) \stackrel{\text{def}}{=} \textbf{if } x = R^j E \textbf{ for some } j \textbf{ then } R^{j+1}E \textbf{ else } x \textbf{ endif}$$

$$UnR(x) \stackrel{\text{def}}{=} \textbf{if } x = R^{j+1} E \textbf{ for some } j \textbf{ then } R^{j}E \textbf{ else } x \textbf{ endif}$$

Thus, values v could be passed with an extra (say, highest order) bit denoting whether the word actually stands for a data value or for $R^v(E)$. R and UnR would then become increment and decrement operations conditional on the highest bit.

If R and UnR are applied to all values at every round, perhaps as unconditional increment and decrement operations, then intermediate error values such as $R(R(E))$ may coincide with valid data values. The algorithm remains correct because UnR (decrement) is always applied to the output of the majority vote.

Both of these implementations of R and UnR require unbounded integers in order to truly satisfy the requirements on R and UnR (for all v, $R(v) \neq E$, and $UnR(R(v)) = v$). However, for an m round OMH, just $m+1$ error values (E up to $R^m(E)$) suffice with suitable modifications to the algorithm.

One could add a comparison of the number of applications of R with the depth of recursion in the algorithm OMH. (Simply computing $R^x(E)$ where x is taken modulo the total number of rounds leads to erroneous results.) Any values with more R's than elapsed rounds may correctly be considered to indicate manifest faults and treated as E, thus reducing the number of possible error values to one more than the number of rounds. In the common case of one-round OMH, two error values, corresponding to E and $R(E)$ suffice. With only a small set of error values, it may no longer be necessary to distinguish them by setting a special bit: they could simply be allocated to values beyond the valid data range.

Using these techniques, one may reduce the overhead of using OMH-like algorithms (as compared to OM) to a small constant number of extra data values, and a slightly more complex algorithm. These implementation techniques have not been formally verified.

Chapter 4

The Formal Specification and Verification

We have formally specified the OMH(n) algorithm and formally verified that it satisfies the properties of agreement and validity using the PVS verification system [22]. The specification language of PVS is a higher-order logic with a very rich type system. This allowed us to specify the OMH algorithm, its assumptions, and properties fairly directly. PVS's theorem prover or proof checker (we use either term, though the latter is more correct) is interactive and operates under the direct control of user: the user chooses each step that is to be applied and PVS performs it, displays the result, and then waits for the next command. PVS differs from most other interactive theorem provers in the power of its basic steps: these can invoke decision procedures for arithmetic, automatic rewriting, induction, and other relatively large units of deduction; it differs from other highly automated theorem provers in being directly controlled by the user. This style of mechanized proof checking allowed us to discover the flaws in our early formulations of the OMH algorithm, and to verify the properties of the final version with relatively little effort.

We describe the formal specification in the next section, and its formal verification in the section after that.

4.1 Formal Specification

The formal specification is a single PVS theory omh (shown starting on page 31) that takes several parameters, beginning with a natural number m denoting the number of rounds of message exchanges to be performed, and a strictly positive natural number n that denotes the number of participants (i.e., channels or "fault

containment units"). Both the natural numbers $(0, 1, 2, \ldots)$ and the strictly positive natural numbers $(1, 2, 3, \ldots)$ are predefined types in PVS (*nat*, and *posnat*, respectively), which are specified in the "prelude" of standard definitions that are automatically loaded into PVS. The prelude theories are described in the PVS language reference [23], and can also be examined on-line using the PVS commands *view-prelude-file* and *view-prelude-theory*.

The parameter list continues by introducing an uninterpreted type T, to represent the class of values exchanged in the algorithm, and an uninterpreted constant *error* used to represent values that are recognized as manifestly erroneous.

The remaining parameters to the OMH theory are the functions R and UnR, representing the "wrapping" and "unwrapping" functions that are performed on values as they are exchanged on the OMH algorithm. These functions must satisfy certain constraints (namely, wrapped values must not look like error values, and unwrapping a wrapped value must return the original value) that are stated as assumptions on the theory OMH, and discussed in Section 3.2. Formally, both are functions from T to T. The function R is used to prevent a value from being discarded by the hybrid majority vote. UnR is used to recover the correct value after the vote. Recall that error values are recorded as the values "sent" by manifest-faulty processors. The first assumption states that no R value is an error.

$$R(t) \neq error.$$

The second assumption states that UnR of R of a value is the same value.

$$UnR(R(t)) = t.$$

The algorithm proceeds through a number of "rounds" counted by the natural numbers $0, 1, \ldots, m$; this range of numbers is specified as the type *rounds*, using the predefined type-constructor *upto* from the PVS prelude. Processors, or "fault containment units" are represented by the natural numbers $0, 1, \ldots, n-1$. This type, called *fcu*, is specified in terms of the predefined type-constructor *below* from the PVS prelude.[1] The type *fcuset* represents sets of *fcu*s, and is specified in terms of the predefined type-constructor *setof*, also from the PVS prelude. Finally, the type *fcuvector* is specified as the type of functions from *fcu*s to T.

Several variables are then introduced, and instantiations of some prelude theories are imported. Prelude theories are always available and do not need to be imported explicitly; the advantage of doing so, however, is that the required instances can be indicated, so that later references can use simple, rather than qualified, names.

[1] A slightly more elegant approach would make *fcu* a type parameter, with an assumption that it is bijective with *below*$[n]$; several prelude theories use this approach.

The theory *finite_cardinality* is one of several cardinality theories available in the PVS prelude; these theories differ in their assumptions concerning the type of the elements of the sets concerned. The theory *finite_cardinality* is applicable to sets whose elements are drawn from a finite type; its parameters are the type concerned (here *fcu*), a natural number that is the cardinality of that type (here n), and a bijection (here *identity*[*fcu*]) from the canonical set *below*[n] of cardinality n to the type concerned.

The theory *filters* defines a function *filter* that returns the set of members of a given set that satisfy a given predicate. Since predicates and sets are equivalent in higher-order logic, this operation is the same as set intersection.[2] The theory *card_set* provides some standard lemmas concerning cardinality, and filters (for example, the cardinality of a set is nonzero if and only if the set is nonempty); it takes the same arguments as *finite_cardinality*. The imported theory *hybridmjrty* is not essential to the main development and is described in Appendix A.

The type *statuses* is defined to be an enumeration of exactly four constants, corresponding to the four categories of behavior: *arbitrary*, *symmetric*, *manifest*, and *good*.[3] The function *status* returns the status of a given processor (or *fcu*); this implicitly enforces our notion that a processor not change status during execution of the agreement protocol. A processor that, in reality, is symmetric-faulty one moment and manifest-faulty the next must be modeled as one that is arbitrary-faulty throughout the computation.

Some shorthands are then defined for describing statuses: a, s, c, and g are predicates recognizing the arbitrary-faulty, symmetric-faulty, manifest-faulty, and good processors, respectively. Similarly, given a set *caucus*, *as*(*caucus*) is the set of arbitrary-faulty processors in *caucus*. The functions *ss*, *cs* and *gs* similarly select the symmetric-faulty, manifest-faulty, and good processors, respectively. A simple lemma, *fincard_all*, states that the cardinality of a set of processors is equal to the sum of the cardinalities of the subsets of its processors of each status. This lemma follows from a property implicit in the definition of statuses as an enumeration type: the members of the enumeration are inclusive and disjoint.

The function *send* captures the properties of sending values from one processor to another. This function takes a value to be sent, a sender, and a receiver as arguments; it returns the value that *would be* received if the receiver were a good processor. The result actually received is irrelevant if the receiver is not a good processor (because the values passed on by faulty receivers are not assumed to be related to those received). We axiomatize the behavior of *send* according to the

[2]The theory *filters* also provides a similar function on lists, which is rather more complex.

[3]Enumeration constants are also overloaded as recognizer functions in PVS. Thus, if s is a variable of type *statuses*, s = *arbitrary* and *arbitrary*(s) are equivalent formulas. The latter form is used in this specification.

status of the sender. The first axiom simply says that a good processor sends correct values to all (good) receivers:

$$g(p) \supset send(t, p, q) = t.$$

Note that here, and in further formal definitions, free variables are universally bound at the outermost level, and the types of all variables are omitted for brevity. See the complete specification for subsidiary and variable declarations. The second axiom says that a manifest-faulty processor always delivers values that are recognized as erroneous by good receivers:

$$c(p) \supset send(t, p, q) = error.$$

The third axiom says that a symmetric-faulty processor sends the same value to all good receivers, although that value is otherwise unconstrained (i.e., it may be any possible value, including those that are recognized as erroneous)

$$s(p) \supset send(t, p, q) = send(t, p, z).$$

Nothing is specified for the behavior of arbitrary-faulty senders. A lemma (called *send5*) is stated and proved that all good receivers obtain the same value when the sender has any status but arbitrary-faulty:

$$\neg a(p) \supset send(t, p, q) = send(t, p, z).$$

A deficiency of this specification is that, because *send* is a function, even arbitrarily faulty processors are consistent from one round to the next: the value $send(t, p, q)$ is some fixed value, suggesting that a faulty processor p, given the same value t, will always send the same (possibly bad) value to the processor q—even in different rounds of the protocol. This fact is not exploited in the proof, but it is not self-evident that this is so. In our verification of the OM algorithm [27], we added the round number as an additional argument to *send* in order to lessen this concern. However, the only way to allay such doubts absolutely is to specify *send* as a relation. Our colleague Shankar has axiomatized the OM algorithm using a relational *send*, and has proven the corresponding correctness conditions. Unfortunately, the relational *send* complicates and obscures the specification (since it forces other functions to become relations also), so we have chosen to retain a functional *send* for this exercise. It is probable that a relational version could be created without great effort.

Our formal specification of OMH is based on our earlier specification of the classical OM algorithm [27]. Rather than simply present the formal specification of OMH as a *fait accompli*, we first reproduce some of the development of the

specification for the OM algorithm from our earlier report, and then transform it into the OMH algorithm.

We start by considering the Interactive-Consistency version of OM, which we call OMIC. We specify OMIC as a function of three arguments: m the number of rounds, v an *fcuvector* giving the private values of each processor, and *caucus* the set of processors participating in (this round of) the algorithm. OMIC will return a "vector" of *fcuvectors*: that is a function from *fcu* to *fcuvector*. Thus $\mathrm{OMIC}(m, v, \mathit{caucus})(p)$ will be the *fcuvector* of processor p following the OMIC algorithm, and $\mathrm{OMIC}(m, v, \mathit{caucus})(p)(q)$ will be p's opinion of q's private value. Notice that we are using higher-order functions here (i.e., functions whose values are functions).

In preparation for formally specifying OMIC, we first state its behavior for the case $m = 0$.

$$\mathrm{OMIC}(0, v, \mathrm{caucus})(p)(q) \;=\; \mathrm{send}(v(q), q, p)$$

Our requirement on OMIC in the case $m = 0$ simply states that p's opinion of q's private value $v(q)$ following the algorithm should be $\mathit{send}(v(q), q, p)$. It might seem that we should require that both p and q should be members of the set *caucus* (as we did in the specification in [27]), but this is unnecessary because the value of the function is irrelevant p or q are not members of *caucus*.

For the case $m = r$, $r > 0$, we require that p's opinion of q's private value should be $\mathit{send}(v(q), q, q)$ if $p = q$,[4] otherwise it should be the majority value in p's *fcuvector*, after performing OMIC with $m = r - 1$ on the current set of processors with q excluded, and the values received from q as the private values. Now the value received by an arbitrary processors z from q is $\mathit{send}(v(q), q, z)$, so the *fcuvector* of such values is

$$(\lambda z : \mathrm{send}(v(q), q, z)).$$

The inner round of OMIC is therefore described by

$$\mathrm{OMIC}(r - 1, (\lambda z : \mathrm{send}(v(q), q, z)), \mathrm{caucus} - \{q\}),$$

and the *fcuvector* received by p following this is

$$\mathrm{OMIC}(r - 1, (\lambda z : \mathrm{send}(v(q), q, z)), \mathrm{caucus} - \{q\})(p).$$

[4]We could specify $v(q)$ in this case; we have chosen the weaker assumption that a faulty processor may not even know its own value.

Thus the required specification is:

```
r > 0
  ⊃ OMIC(r, v, caucus)(p)(q)
    = IF p = q THEN send(v(q), q, q)
  ELSE
    Majority(caucus − {q},
             OMIC(r − 1, (λ z : send(v(q), q, z)), caucus − {q})(p))
  ENDIF
```

The function *Majority* takes a set of processors (here *caucus*$-\{q\}$), and an *fcuvector*, and computes the majority value (if any) in that vector over that set.[5]

The two behaviors stated above (for the cases $m = 0$, and $m > 0$, respectively) could be specified as axioms defining the function OMIC; we prefer, however, to specify the function definitionally and to deduce those properties as (straightforward) lemmas. The advantage of the definitional specification is that the PVS typechecker will guarantee its soundness (in the sense of not introducing inconsistencies). To do this, we are required to exhibit a *measure* function that takes the same arguments as OMIC and whose value is a natural number that can be proved to decrease across recursive calls. In the present case, we use the function that returns the round number as the measure function. The final specification for OMIC is given below.

```
OMIC(r, v, caucus) : RECURSIVE [fcu → fcuvector] =
  IF r = 0 THEN (λ p : (λ q : send(v(q), q, p)))
  ELSE
   (λ p :
     (λ q :
       IF p = q THEN send(v(q), q, q)
       ELSE
         Majority(caucus − {q},
                  OMIC(r − 1, (λ z : send(v(q), q, z)),
                       caucus − {q})(p))
       ENDIF))
  ENDIF
 MEASURE (λ r, v, caucus → nat : r)
```

[5] Requiring this function to be implemented by a majority vote overspecifies the problem. All that is really required is that if the good processors form a majority in *caucus*, and if all the good processors have the same value in the vector, then that is the value of the *Majority* function. Taking the median of the values of the members of *caucus* (assuming they come from an ordered set) would also satisfy this specification (as was correctly noted by Lamport, Shostak and Pease [17, page 388]).

The next step is to convert our specification of the basic Oral Messages algorithm from the Interactive-Consistency to the Byzantine Generals formulation. We specify the Byzantine Generals form by a function OMBG that is similar to OMIC, but takes an additional (first) argument giving the identity of the Transmitter (or Commanding General), and replaces the *fcuvector* of private values by a single private value (that of the Transmitter). The result returned by OMBG is a simple *fcuvector*:

$$\text{OMBG}(G, m, t, \text{caucus})(p)$$

is processor p's opinion of the Transmitter G's private value t following an m-round exchange. If we assume that OMIC is available, then the behavior required of OMBG in the case $r > 0$ can be derived directly from that of OMIC:

```
r > 0
  ⊃ OMBG(G, r, t, caucus)(p)
   = IF p = G THEN send(t, G, p)
  ELSE
   Majority(caucus - {q},
              OMIC(r - 1, (λ z : send(t, G, z)), caucus - {q})(p))
  ENDIF
```

The next step is to replace the inner call to OMIC by one to OMBG. Now

$$\text{OMIC}(r-1, (\lambda z : \text{send}(t, G, z)), \text{caucus} - \{q\})(p)$$

is an *fcuvector* giving processor p's opinion of the values received by each processor when G sends them t. Using OMBG, p's opinion of the value received by processor z in this circumstance can be written

$$\text{OMBG}(z, r-1, \text{send}(t, G, z), \text{caucus} - \{q\})(p)$$

(i.e., z takes the part of the Transmitter, distributing the value *send*(t, G, z) received from the "real" Transmitter). Thus, the required *fcuvector* giving p's opinion of the values received by all such processors z is given by:

$$(\lambda z : \text{OMBG}(z, r-1, \text{send}(t, G, z), \text{caucus} - \{q\})(p)).$$

In this way we arrive at the specification for OMBG shown below.

```
OMBG(G, r, t, caucus) : RECURSIVE fcuvector =
  IF r = 0 THEN (λ p : send(t, G, p))
  ELSE
   (λ p :
     IF p = G THEN send(t, G, p)
     ELSE
       Majority(caucus − {G},
                 (λ z : OMBG(z, r − 1, send(t, G, z), caucus − {G})(p)))
     ENDIF)
  ENDIF
 MEASURE (λ G, r, t, caucus → nat : r)
```

The final step is to transform this specification into one for OMH. The only differences between OMBG and OMH are that the latter uses *HybridMajority* instead of the simple *Majority* function, and "wraps" and "unwraps" the values sent and received in the recursive calls with the functions *R* and *UnR*, respectively. Thus the specification for OMH given below is easily derived.

```
OMH(G, r, t, caucus) : RECURSIVE fcuvector =
  IF r = 0 THEN (λ p : send(t, G, p))
  ELSE
   (λ p :
     IF p = G THEN send(t, G, p)
     ELSE
       UnR(HybridMajority(caucus − {G},
                         (λ z : OMH(z, r − 1, R(send(t, G, z)), caucus − {G})(p))))
     ENDIF)
  ENDIF
 MEASURE (λ G, r, t, caucus → nat : r)
```

It remains to specify the properties required of the functions *HybridMajority*, *UnR*, and *R*. The function *HybridMajority* is intended to be like the previous (standard) *Majority* function, except that all *error* values are excluded. Recall from our earlier discussion that the *Majority* function does not actually need to be a majority vote, so it is preferable to specify the properties required of it axiomatically. The same is true of *HybridMajority*, which is axiomatized below. Although these two properties are all that is required of an implementation of *HybridMajority*, we have provided a concrete implementation of *HybridMajority* based on the Boyer-Moore MJRTY algorithm [3], and proved that the axioms below are satisfied by this implementation. Thus the following may be considered axioms, or may be considered lemmas proven by appeal to the ***hybridmjrty*** theory described in Appendix A.

$$
\begin{aligned}
&|\text{gs(caucus)}| \;>\; |\text{as(caucus)}| \;+\; |\text{ss(caucus)}| \\
&\quad \wedge\; (\forall\, p :\; g(p) \;\wedge\; p \in \text{caucus} \;\supset\; v(p) \;=\; t) \\
&\quad\; \wedge\; t \;\neq\; \text{error} \;\wedge\; (\forall\, p :\; c(p) \;\wedge\; p \in \text{caucus} \;\supset\; v(p) \;=\; \text{error}) \\
&\quad \supset\; \text{HybridMajority}(\text{caucus}, v) \;=\; t
\end{aligned}
$$

The antecedent to the implication in this specification is complicated, but can be read as follows. The function *HybridMajority* takes two arguments, a set of processors (i.e., an *fcuset*), which we call the *caucus*, and a vector mapping processors to values (i.e., an *fcuvector*). If the vector records the same value for all good processors in the caucus, and the vector records an error value for all manifest-faulty processors in the caucus, and there are more good processors in the caucus than the sum of arbitrary-faulty and symmetric-faulty processors in the caucus, then *HybridMajority* returns the same value as that recorded in the vector for the good processors. Any implementation of *HybridMajority* that does in fact compute the true majority after casting out error values would satisfy this axiom.

The second axiom states that the value returned depends only on the values recorded in the vector for the processors in the caucus. Although *HybridMajority* is a function, it could potentially be implemented in such a way that when there is no majority (i.e., when the antecedent to the implication above is false), the output depends on values of the vector corresponding to processors not in the caucus, or other irrelevant information contained in the arguments. The second axiom prohibits this kind of behavior.

$$
\begin{aligned}
&(\forall\, p :\; p \in \text{caucus} \;\supset\; v_1(p) \;=\; v_2(p)) \\
&\quad \supset\; \text{HybridMajority}(\text{caucus}, v_1) \;=\; \text{HybridMajority}(\text{caucus}, v_2)
\end{aligned}
$$

The remainder of the specification consists of interesting properties of the OMH algorithm. Many of the following theorems are first defined as predicates, then a lemma asserting that this predicate is universal is proved by induction, and then a theorem giving the result in the form desired is derived from the lemma. This style of breaking a specification into a predicate and a separate lemma and theorem is quite useful in formal systems. Many other large specifications use this technique [30,37].

Note that in the semiformal specification there was a notion of the *value actually sent* by the transmitter. This is very close to the value of the function *send*(t,p,q), although we have axiomatized *send* so that manifest-faulty processors "send" error. Thus the formal specification of *Validity* uses *send* where the semiformal specification uses case analysis and the notion of the value actually sent by symmetric-faulty processors.

The first big property is *Validity*, stating that if the transmitter is not arbitrary-faulty, then this algorithm achieves the same result as *send*. This captures the correct behavior when the transmitter is good (send delivers the correct value), symmetric-faulty (*send* delivers the same wrong value to all receivers, which then agree on this value), and manifest-faulty (*send* effectively delivers the value (*error* to all receivers which then agree on *error* as the transmitter's value).

$$
\begin{array}{l}
\neg\, a(q) \\
\quad \wedge\ p \in \text{caucus} \\
\quad\quad \wedge\ q \in \text{caucus} \\
\quad\quad\quad \wedge\ |\text{caucus}| > 2 \times (|\text{as}(\text{caucus})| + |\text{ss}(\text{caucus})|) + |\text{cs}(\text{caucus})| + r \\
\quad \supset\ \text{OMH}(q, r, t, \text{caucus})(p) = \text{send}(t, q, p)
\end{array}
$$

The next property is *Agreement*, which states that if two receivers are both good they will agree, whatever the status of the transmitter.

$$
\begin{array}{l}
g(p) \\
\quad \wedge\ g(q) \\
\quad\quad \wedge\ p \in \text{caucus} \\
\quad\quad\quad \wedge\ q \in \text{caucus} \\
\quad\quad\quad\quad \wedge\ z \in \text{caucus} \\
\quad\quad\quad\quad\quad \wedge\ |\text{caucus}| > 2 \times (|\text{as}(\text{caucus})| + |\text{ss}(\text{caucus})|) + |\text{cs}(\text{caucus})| + r \\
\quad\quad\quad\quad\quad\quad \wedge\ r \geq |\text{as}(\text{caucus})| \\
\quad \supset\ \text{OMH}(z, r, t, \text{caucus})(p) = \text{OMH}(z, r, t, \text{caucus})(q))
\end{array}
$$

The next property, *Validity_final*, instantiates the inductive validity property with the full set of processors. The property *Validity_Corollary* addresses the special case when the transmitter is good. In this case the correct value is agreed upon by all good receivers. The property *Agreement_final* instantiates the inductive agreement property with the full set of processors.

The remainder of the specification addresses the special case when there are no symmetric-faulty nor arbitrary-faulty processors. In this case a somewhat better bound can be achieved with regard to manifest-faulty processors. In particular, the bounds given by the general versions of the theorems are not as good as can be achieved with simple non-Byzantine resilient algorithms. However, as described in Section 3.4, OMH does actually achieve optimal behavior in these cases, as shown by the alternative analysis described below.

The same set of interesting properties described above is then repeated with the added assumption that there are no arbitrary-faulty nor symmetric-faulty processors. The bounds proven in these cases deliver more resilience to manifest faults.

Note that in these analyses, there is at least one good receiver, and the remainder of the receivers are either good or manifest-faulty, so the good receivers will always win the majority vote. A detail is the extra requirement that there be more processors than rounds, since it is difficult to assert properties of the OMH algorithm run on the empty set of processors.

Note that analogous "ArbitraryOnly" theorems also hold, giving optimal bounds,[6] although these bounds are trivial consequences of the general theorem. The analogous "SymmetricOnly" theorems would not give optimal bounds, as was discussed in Section 3.4.

```
omh[m : nat, n : posnat, T : TYPE, error : T, R, UnR : [T → T]] :
 THEORY
  BEGIN

  ASSUMING
   act_ax : ASSUMPTION (∀ (t : T) : R(t) ≠ error)

   unact_ax : ASSUMPTION (∀ (t : T) : UnR(R(t)) = t)

  ENDASSUMING

 rounds : TYPE  = upto[m]
 t : VAR T
 fcu : TYPE  = below[n]
 fcuset : TYPE  = setof[fcu]
 fcuvector : TYPE  = [fcu → T]
 G, p, q, z : VAR fcu
 v, v1, v2 : VAR fcuvector
 caucus : VAR fcuset
 r : VAR rounds
 IMPORTING
  finite_cardinality[fcu, n, identity[fcu]],
  filters[fcu],
  card_set[fcu, n, identity[fcu]],
  hybridmjrty[T, n, error]
```

[6] Pease, Shostak and Lamport [25] proved that at least $3a+1$ processors are required to withstand a arbitrary faults. This result has been formally verified by Bevier and Young [2,1] using the Boyer-Moore prover.

statuses : TYPE $=$ {arbitrary, symmetric, manifest, good}
status : [fcu $\rightarrow$ statuses]
$a(z)$: bool $=$ arbitrary(status(z))

$s(z)$: bool $=$ symmetric(status(z))

$c(z)$: bool $=$ manifest(status(z))

$g(z)$: bool $=$ good(status(z))

as(caucus) : fcuset $=$ filter(caucus, a)

ss(caucus) : fcuset $=$ filter(caucus, s)

cs(caucus) : fcuset $=$ filter(caucus, c)

gs(caucus) : fcuset $=$ filter(caucus, g)

fincard_all :
LEMMA
$|\text{caucus}| = |\text{as(caucus)}| + |\text{ss(caucus)}| + |\text{cs(caucus)}| + |\text{gs(caucus)}|$

send : [T, fcu, fcu $\rightarrow$ T]
send1 : AXIOM $g(p) \supset \text{send}(t, p, q) = t$

send2 : AXIOM $c(p) \supset \text{send}(t, p, q) = \text{error}$

send4 : AXIOM $s(p) \supset \text{send}(t, p, q) = \text{send}(t, p, z)$

send5 : LEMMA $\neg\, a(p) \supset \text{send}(t, p, q) = \text{send}(t, p, z)$

HybridMajority(caucus, v) : T $=$ proj_1(Hybrid_mjrty(caucus, v, n))

HybridMajority_ax1 :
LEMMA
$|\text{gs(caucus)}| > |\text{as(caucus)}| + |\text{ss(caucus)}|$
$\wedge\ (\forall\, p : g(p) \wedge p \in \text{caucus} \supset v(p) = t)$
$\wedge\ t \neq \text{error} \wedge (\forall\, p : c(p) \wedge p \in \text{caucus} \supset v(p) = \text{error})$
$\supset$ HybridMajority(caucus, v) $= t$

HybridMajority_ax2 :
LEMMA
$(\forall\, p : p \in \text{caucus} \supset v_1(p) = v_2(p))$
$\supset$ HybridMajority(caucus, v_1) $=$ HybridMajority(caucus, v_2)

```
OMH(G, r, t, caucus) : RECURSIVE fcuvector =
  IF r = 0 THEN (λ p : send(t, G, p))
  ELSE
   (λ p :
     IF p = G THEN send(t, G, p)
     ELSE
      UnR(HybridMajority(caucus − {G},
                    (λ q : OMH(q, r − 1, R(send(t, G, q)), caucus − {G})(p))))
     ENDIF)
  ENDIF
 MEASURE (λ G, r, t, caucus → nat : r)

Validity_Prop(r) :
 bool
   =
  (∀ p, q, caucus, t :
    ¬ a(q)
      ∧ p ∈ caucus
       ∧ q ∈ caucus
        ∧ |caucus|
          > 2 × (|as(caucus)| + |ss(caucus)|) + |cs(caucus)| + r
      ⊃ OMH(q, r, t, caucus)(p) = send(t, q, p))

Validity : LEMMA Validity_Prop(r)

Agreement_Prop(r) :
 bool
   =
  (∀ p, q, z, caucus, t :
    (g(p)
      ∧ g(q)
       ∧ p ∈ caucus
        ∧ q ∈ caucus
         ∧ z ∈ caucus
          ∧ |caucus|
            > 2 × (|as(caucus)| + |ss(caucus)|) + |cs(caucus)|
             + r
           ∧ r ≥ |as(caucus)|)
     ⊃ OMH(z, r, t, caucus)(p) = OMH(z, r, t, caucus)(q))

Agreement : LEMMA Agreement_Prop(r)
```

Validity_Final :
THEOREM
$$
\begin{aligned}
& g(p) \\
& \quad \wedge \neg a(G) \\
& \quad\quad \wedge |a| \leq m \\
& \quad\quad\quad \wedge 2 \times |a| + 2 \times |s| + |c| + m < n \\
& \quad \supset \mathrm{OMH}(G, m, t, \mathrm{fullset[fcu]})(p) = \mathrm{send}(t, G, p)
\end{aligned}
$$

Validity_Corollary :
THEOREM
$$
\begin{aligned}
& g(p) \\
& \quad \wedge g(G) \\
& \quad\quad \wedge |a| \leq m \\
& \quad\quad\quad \wedge 2 \times |a| + 2 \times |s| + |c| + m < n \\
& \quad \supset \mathrm{OMH}(G, m, t, \mathrm{fullset[fcu]})(p) = t
\end{aligned}
$$

Agreement_Final :
THEOREM
$$
\begin{aligned}
& g(p) \\
& \quad \wedge g(q) \\
& \quad\quad \wedge |a| \leq m \\
& \quad\quad\quad \wedge 2 \times |a| + 2 \times |s| + |c| + m < n \\
& \quad \supset \mathrm{OMH}(G, m, t, \mathrm{fullset[fcu]})(p) = \mathrm{OMH}(G, m, t, \mathrm{fullset[fcu]})(q)
\end{aligned}
$$

Crash_Only_Validity_Prop(r) :
bool
$$
\begin{aligned}
& = \\
& (\forall\, p, q, \mathrm{caucus}, t : \\
& \quad g(p) \\
& \quad\quad \wedge p \in \mathrm{caucus} \\
& \quad\quad\quad \wedge q \in \mathrm{caucus} \\
& \quad\quad\quad\quad \wedge |\mathrm{as(caucus)}| = 0 \wedge |\mathrm{ss(caucus)}| = 0 \wedge |\mathrm{caucus}| > r \\
& \quad\quad \supset \mathrm{OMH}(q, r, t, \mathrm{caucus})(p) = \mathrm{send}(t, q, p))
\end{aligned}
$$

Crash_Only_Validity : LEMMA Crash_Only_Validity_Prop(r)

Crash_Only_Agreement_Prop(r) :
bool
=
($\forall$ p, q, z, caucus, t :
$g(p)$
$\wedge$ $g(q)$
$\wedge$ $p \in$ caucus
$\wedge$ $q \in$ caucus
$\wedge$ $z \in$ caucus
$\wedge$ |as(caucus)| = 0 $\wedge$ |ss(caucus)| = 0 $\wedge$ |caucus| > r
$\supset$ OMH(z, r, t, caucus)(p) = OMH(z, r, t, caucus)(q))

Crash_Only_Agreement : LEMMA Crash_Only_Agreement_Prop(r)

Crash_Only_Validity_Final :
THEOREM
$g(p)$ $\wedge$ $|a| = 0$ $\wedge$ $|s| = 0$ $\wedge$ |fullset[fcu]| > m
$\supset$ OMH(G, m, t, fullset[fcu])(p) = send(t, G, p)

Crash_Only_Validity_Corollary :
THEOREM
$g(p)$ $\wedge$ $g(G)$ $\wedge$ $|a| = 0$ $\wedge$ $|s| = 0$ $\wedge$ |fullset[fcu]| > m
$\supset$ OMH(G, m, t, fullset[fcu])(p) = t

Crash_Only_Agreement_Final :
THEOREM
$g(p)$ $\wedge$ $g(q)$ $\wedge$ $|a| = 0$ $\wedge$ $|s| = 0$ $\wedge$ |fullset[fcu]| > m
$\supset$ OMH(G, m, t, fullset[fcu])(p) = OMH(G, m, t, fullset[fcu])(q)

END omh

4.2 Formal Verification

The formal verifications corresponding Lemma 1 and Theorems 1 and 2 are proved by induction on the number of rounds, and follow the informal proofs quite closely. The theorem prover of PVS with its built-in arithmetic decision procedures and rewriting allowed the formal proof to be constructed at a relatively high level without being mired in detail. The PVS system allows partial proofs to be replayed under alternative assumptions, facilitating the exploration of generalizations and special cases, such as that reported in Theorem 2, formally reflected in the *Crash_Only* variants of the theorems. Another example of this sort of exploration was the removal of the assumption that error values are disjoint from good data values. The proof of each lemma in the specification is described abstractly below.

The first lemma, *fincard_all*, states that the cardinality of an entire set of processors is equal to the sum of the cardinalities of the processors in that set of each status. This lemma follows from properties implicit in the definition of *statuses*: that they are inclusive and disjoint. In detail, the formal proof requires 30 user-supplied steps in PVS, seven of which are **ground** or **assert**, which invoke the ground decision procedures of PVS.

The second lemma, *send5*, states that all non-arbitrary-faulty processors exhibit symmetric sending behavior. Informally, the proof of this property appeals to the fact that the four statuses—*arbitrary*, *symmetric*, *manifest*, and *good*—are inclusive. Case analysis and appeal to the send axioms *send1*, *send2*, and *send4* essentially completes the proof. In two cases, such as that when the transmitter is manifest-faulty, the relevant axiom must be applied twice. The entire formal proof comprises 14 user-supplied steps in the PVS interactive verification system.

The most complicated proof constructed for this specification is for Lemma 1 (called *Validity* in the formal specification), stating that if the transmitter is not arbitrary-faulty then all good receivers end up with the value actually sent by the transmitter. This proof contains 13 invocations of lemmas and axioms, most of them basic lemmas from the prelude and axioms from the OMH theory. The lemmas and axioms cited in the proof are: *induction*; *statuses_inclusive*, and *statuses_disjoint*, which are the automatically-generated assertions that the set of four statuses are inclusive and disjoint; *fincard_remove*, a lemma giving the cardinality of a set after an element has been removed (used often—thirteen times); *fincard_filter*, a lemma asserting that the cardinality of a set is not less than the cardinality of that set with some elements removed; *fincard_all*, described above; *send5*, described above; *remove_comm*, a lemma asserting that the order in which elements are removed from a set is immaterial, used four times; and axioms (such as the definitional axioms for *send*) brought into the proof explicitly six times. The entire proof consists of 80 user-suggested steps, 15 of which are PVS **assert** commands, which invoke rather powerful decision procedures for ground arithmetic [31,32]. After some experimentation with alternative specifications, including constructing partial failed proofs of this lemma for alternative versions of the algorithm OMH, the first proof of *Validity* was constructed from scratch in a few hours.

The proof of the crash-only variant of *Validity* is very similar in nature to the general version of validity, although it contains one fewer user-supplied steps, 14 of which are **assert**.

The proof of the *Agreement* property contains 10 invocations of lemmas and axioms, and consists of 73 steps, including 13 calls to the ground decision procedures. This proof was constructed from scratch in a few hours.

Table 4.1 summarizes some gross measures of the size and difficulty of constructing proofs for the lemmas and theorems of this specification. The first column is

formula name	user-supplied steps	number of inductions	uses of assert
fincard_all	30	1	7
send5	14	0	1
Validity	80	1	15
Agreement	73	1	13
Validity_final	36	0	4
Validity_Cor	14	0	2
Agreement_final	51	0	4
Crash_Only_Validity	79	1	14
Crash_Only_Agreement	41	1	6
Crash_Only_Valdity_final	29	0	0
Crash_Only_Validity_Cor	8	0	1
Crash_Only_Agreement_final	21	0	0
Hybridmajority_ax1	78	2	16
Hybridmajority_ax2	20	1	3

Table 4.1: Statistics for the Proofs Performed

the name of the formula concerned. The second column is the total number of user-suggested proof steps in the final proof. The third column counts the uses of induction. The fourth column counts the uses of **ground** or **assert**, which invoke the ground decision procedures; these roughly correspond to the number of significant branches in a proof.

The critical measure, however, for specification and verification tasks is not the size of individual proofs, but the total time taken from problem understanding through complete formal proof. The effort reported here took less than a month of part time work, including the exploration of flawed modifications to Algorithm Z that seemed informally plausible, and a change in notation for expository purposes. Producing this report took far more time than the formal specification and verification combined.

Full machine-readable PVS specifications and PVS proofs of the entire proof chain are available from the authors.

4.2.1 Portion of PVS Proof of Validity

In order to give an idea of the formal proof as interactively developed using PVS, we reproduce a prettyprinted version of such a proof (slightly edited for readability). It begins with the theorem name, `Validity`, and the initial sequent. Sequents are presented as a list of numbered hypotheses, a horizontal line, and a list of numbered conclusions. One may read a sequent as stating that the conjunction of the hypotheses implies the disjunction of the conclusions. Initially, there are no hypotheses, and only one conclusion, stating that for any number of rounds, the inductive validity property holds for that many rounds. The first step in the proof is the application of induction on the number of rounds. This leads to two subgoals, called `Validity.1` and `Validity.2`. For `Validity.1`, the base case of the induction, the definition of the inductive validity property is expanded and the result is skolemized (fresh constants are introduced in place of universally quantified variables). Then the definition of OMH is expanded and reduced. (Recall the specification given earlier; zero-round OMH reduces to *send*.) This completes the branch of the proof corresponding to the base case of the induction.

The remaining branch, called `Validity.2`, is then proved. This branch requires it to be shown that for any number of rounds r, if the inductive validity property holds for r rounds, then it also holds for $r + 1$ rounds. By skolemizing, expanding definitions, and applying propositional simplification, we arrive at the crux of the proof. Here we must show that if OMH behaves correctly at r rounds, then the *UnR* of the *HybridMajority* of the result of all other receivers utilizing OMH to broadcast R of the value they received from the transmitter is the same as the value actually sent by the transmitter. This must be demonstrated under certain other assumptions, such as that are enough nonfaulty processors, and that the transmitter is not arbitrary faulty. The proof proceeds by utilizing a property of *HybridMajority*, called *HybridMajority_ax1*. There are four hypotheses of this property, and one conclusion. After quantifying appropriately, the proof is split into five cases, corresponding to a proof of each hypothesis of *HybridMajority_ax1* and a proof from the conclusion of *HybridMajority_ax1* to the conclusion of the actual property of interest. The proof of the latter (`Validity.2.1`) proceeds by bringing in the assumption *unact_ax*, hiding some irrelevant formulas, and invoking the decision procedures.

`Validity:`

$$\frac{}{\{1\}\quad (\forall\, (r: \text{rounds}): \text{Validity_Prop}(r))}$$

Inducting on r yields 2 subgoals: `Validity.1` and `Validity.2`

`Validity.1:`

$$\frac{}{\{1\}\quad \text{Validity_Prop}(0)}$$

Expanding the definition of Validity_Prop

`Validity.1:`

$$\begin{array}{ll} \hline \{1\} & (\forall\, (p, q: \text{fcu}), (\text{caucus}: \text{fcuset}), (t: T): \\ & \quad \neg\, \text{arbitrary}(\text{status}(q)) \\ & \quad\quad \wedge\ \text{caucus}(p) \\ & \quad\quad\quad \wedge\ \text{caucus}(q) \\ & \quad\quad\quad\quad \wedge\ |\text{caucus}| \\ & \quad\quad\quad\quad\quad > 2 \times |\text{filter}(\text{caucus}, a)| + 2 \times |\text{filter}(\text{caucus}, s)| \\ & \quad\quad\quad\quad\quad\quad + |\text{filter}(\text{caucus}, c)| \\ & \quad\quad\quad\quad\quad\quad + 0 \\ & \quad\quad \supset \text{OMH}(q, 0, t, \text{caucus})(p) = \text{send}(t, q, p)) \end{array}$$

For the top quantifier in 1, we introduce Skolem constants: $(p', q', caucus', t')$ and apply disjunctive simplification to flatten the sequent,

`Validity.1:`

$$\begin{array}{ll} \{-1\} & \text{caucus}'(p') \\ \{-2\} & \text{caucus}'(q') \\ \{-3\} & |\text{caucus}'| \\ & \quad > 2 \times |\text{filter}(\text{caucus}', a)| + 2 \times |\text{filter}(\text{caucus}', s)| \\ & \quad\quad + |\text{filter}(\text{caucus}', c)| \\ & \quad\quad + 0 \\ \hline \{1\} & \text{arbitrary}(\text{status}(q')) \\ \{2\} & \text{OMH}(q', 0, t', \text{caucus}')(p') = \text{send}(t', q', p') \end{array}$$

Expanding the definition of OMH completes the proof of `Validity.1`.

`Validity.2:`

$\{1\}$ $(\forall\ (r : \text{upto}[m]) : r < m \land \text{Validity_Prop}(r) \supset \text{Validity_Prop}(r + 1))$

For the top quantifier in 1, we introduce Skolem constants: (r') and apply disjunctive simplification to flatten the sequent,

`Validity.2:`

$\{-1\}$ $r' < m$

$\{-2\}$ $\text{Validity_Prop}(r')$

$\{1\}$ $\text{Validity_Prop}(r' + 1)$

Expanding the definition of Validity_Prop in formula 1

`Validity.2:`

$\{-1\}$ $r' < m$

$\{-2\}$ $\text{Validity_Prop}(r')$

$\{1\}$

$$
\begin{aligned}
&(\forall\ (p, q : \text{fcu}), (\text{caucus} : \text{fcuset}), (t : T) : \\
&\quad \neg\ \text{arbitrary}(\text{status}(q)) \\
&\qquad \land\ \text{caucus}(p) \\
&\qquad\quad \land\ \text{caucus}(q) \\
&\qquad\qquad \land\ |\text{caucus}| \\
&\qquad\qquad\quad > \\
&\qquad\qquad 2 \times |\text{filter}(\text{caucus}, a)| + 2 \times |\text{filter}(\text{caucus}, s)| \\
&\qquad\qquad\quad + |\text{filter}(\text{caucus}, c)| \\
&\qquad\qquad\quad + r' \\
&\qquad\qquad\quad + 1 \\
&\qquad \supset \text{OMH}(q, r' + 1, t, \text{caucus})(p) = \text{send}(t, q, p))
\end{aligned}
$$

For the top quantifier in 1, we introduce Skolem constants: $(p', q', caucus', t')$ and apply disjunctive simplification to flatten the sequent,

`Validity.2:`

$\{-1\}$ $r' < m$

$\{-2\}$ Validity_Prop(r')

$\{-3\}$ caucus$'(p')$

$\{-4\}$ caucus$'(q')$

$\{-5\}$ $|\text{caucus}'| > 2 \times |\text{filter}(\text{caucus}', a)| + 2 \times |\text{filter}(\text{caucus}', s)| + |\text{filter}(\text{caucus}', c)| + r' + 1$

$\{1\}$ arbitrary(status(q'))

$\{2\}$ $\text{OMH}(q', r' + 1, t', \text{caucus}')(p') = \text{send}(t', q', p')$

Expanding the definition of OMH

`Validity.2:`

$\{-1\}$ $r' < m$

$\{-2\}$ Validity_Prop(r')

$\{-3\}$ caucus$'(p')$

$\{-4\}$ caucus$'(q')$

$\{-5\}$ $|\text{caucus}'|$
$>$
$2 \times |\text{filter}(\text{caucus}', a)| + 2 \times |\text{filter}(\text{caucus}', s)|$
$+ |\text{filter}(\text{caucus}', c)|$
$+ r'$
$+ 1$

$\{1\}$ arbitrary(status(q'))

$\{2\}$ IF $p' = q'$ THEN send(t', q', p')
ELSE
UnR(HybridMajority(caucus$' - \{q'\}$,
$(\lambda\ (q : \text{fcu}) : \text{OMH}(q, r', R(\text{send}(t', q', q)), \text{caucus}' - \{q'\})(p'))))$
ENDIF
$= \text{send}(t', q', p')$

Lifting IF-conditions to the top level,
and by propositional simplification,

`Validity.2:`

{-1} $r' < m$

{-2} Validity_Prop(r')

{-3} caucus'(p')

{-4} caucus'(q')

{-5} |caucus'|
$>$
$2 \times |\text{filter}(\text{caucus}', a)| + 2 \times |\text{filter}(\text{caucus}', s)|$
$+ |\text{filter}(\text{caucus}', c)|$
$+ r'$
$+ 1$

{1} $p' = q'$

{2} UnR(HybridMajority(caucus' $- \{q'\}$,
$(\lambda\ (q : \text{fcu}) : \text{OMH}(q, r', R(\text{send}(t', q', q)), \text{caucus}' - \{q'\})(p'))))$
$= \text{send}(t', q', p')$

{3} arbitrary(status(q'))

Applying HybridMajority_ax1

`Validity.2:`

$\{-1\}$ $(\forall\ (\text{caucus}: \text{fcuset}), (t: T), (v: \text{fcuvector}):$
$|\text{gs}(\text{caucus})| > |\text{as}(\text{caucus})| + |\text{ss}(\text{caucus})|$
$\wedge\ (\forall\ (p: \text{fcu}): g(p) \wedge p \in \text{caucus} \supset v(p) = t)$
$\wedge\ t \neq \text{error} \wedge (\forall\ (p: \text{fcu}): c(p) \wedge p \in \text{caucus} \supset v(p) = \text{error})$
$\supset \text{HybridMajority}(\text{caucus}, v) = t)$

$\{-2\}$ $r' < m$

$\{-3\}$ $\text{Validity_Prop}(r')$

$\{-4\}$ $\text{caucus}'(p')$

$\{-5\}$ $\text{caucus}'(q')$

$\{-6\}$ $|\text{caucus}'|$
$>$
$2 \times |\text{filter}(\text{caucus}', a)| + 2 \times |\text{filter}(\text{caucus}', s)|$
$+ |\text{filter}(\text{caucus}', c)|$
$+ r'$
$+ 1$

$\{1\}$ $p' = q'$

$\{2\}$ $\text{UnR}(\text{HybridMajority}(\text{caucus}' - \{q'\},$
$(\lambda\ (q: \text{fcu}): \text{OMH}(q, r', R(\text{send}(t', q', q)), \text{caucus}' - \{q'\})(p'))))$
$= \text{send}(t', q', p')$

$\{3\}$ $\text{arbitrary}(\text{status}(q'))$

Instantiating the top quantifier in -1 with the terms:
$\text{remove}(q', \text{caucus}')$
$\text{R}(\text{send}(t', q', p'))$
$(\lambda(q: fcu): \text{OMH}(q, r', \text{R}(\text{send}(t', q', q)), \text{remove}(q', \text{caucus}'))(p'))$

```
Validity.2:
```

$\{-1\}$ $|\mathrm{gs}(\mathrm{caucus}' - \{q'\})|$
$> |\mathrm{as}(\mathrm{caucus}' - \{q'\})| + |\mathrm{ss}(\mathrm{caucus}' - \{q'\})|$
$\wedge$
$(\forall\ (p : \mathrm{fcu}) :$
$g(p) \wedge p \in \mathrm{caucus}' - \{q'\}$
$\supset$
$(\lambda\ (q : \mathrm{fcu}) :$
$\mathrm{OMH}(q, r', R(\mathrm{send}(t', q', q)), \mathrm{caucus}' - \{q'\})(p'))(p)$
$= R(\mathrm{send}(t', q', p')))$
$\wedge\ R(\mathrm{send}(t', q', p')) \neq \mathrm{error}$
$\wedge$
$(\forall\ (p : \mathrm{fcu}) :$
$c(p) \wedge p \in \mathrm{caucus}' - \{q'\}$
$\supset$
$(\lambda\ (q : \mathrm{fcu}) :$
$\mathrm{OMH}(q, r', R(\mathrm{send}(t', q', q)),$
$\mathrm{caucus}' - \{q'\})(p'))(p)$
$= \mathrm{error})$
$\supset$
$\mathrm{HybridMajority}(\mathrm{caucus}' - \{q'\},$
$(\lambda\ (q : \mathrm{fcu}) :$
$\mathrm{OMH}(q, r', R(\mathrm{send}(t', q', q)),$
$\mathrm{caucus}' - \{q'\})(p')))$
$= R(\mathrm{send}(t', q', p'))$

$\{-2\}$ $r' < m$

$\{-3\}$ $\mathrm{Validity_Prop}(r')$

$\{-4\}$ $\mathrm{caucus}'(p')$

$\{-5\}$ $\mathrm{caucus}'(q')$

$\{-6\}$ $|\mathrm{caucus}'|$
$>$
$2 \times |\mathrm{filter}(\mathrm{caucus}', a)| + 2 \times |\mathrm{filter}(\mathrm{caucus}', s)|$
$+ |\mathrm{filter}(\mathrm{caucus}', c)|$
$+ r'$
$+ 1$

$\{1\}$ $p' = q'$

$\{2\}$ $\mathrm{UnR}(\mathrm{HybridMajority}(\mathrm{caucus}' - \{q'\},$
$(\lambda\ (q : \mathrm{fcu}) : \mathrm{OMH}(q, r', R(\mathrm{send}(t', q', q)), \mathrm{caucus}' - \{q'\})(p'))))$
$= \mathrm{send}(t', q', p')$

$\{3\}$ $\mathrm{arbitrary}(\mathrm{status}(q'))$

Splitting conjunctions yields 5 subgoals:

Validity.2.1:

$\{-1\}$ HybridMajority(caucus′ − $\{q'\}$,
(λ (q : fcu) :
OMH(q, r', R(send(t', q', q)),
caucus′ − $\{q'\}$)(p')))
= R(send(t', q', p'))

$\{-2\}$ $r' < m$

$\{-3\}$ Validity_Prop(r')

$\{-4\}$ caucus′(p')

$\{-5\}$ caucus′(q')

$\{-6\}$ |caucus′|
>
2 × |filter(caucus′, a)| + 2 × |filter(caucus′, s)|
+ |filter(caucus′, c)|
+ r'
+ 1

$\{1\}$ $p' = q'$

$\{2\}$ UnR(HybridMajority(caucus′ − $\{q'\}$,
(λ (q : fcu) : OMH(q, r', R(send(t', q', q)), caucus′ − $\{q'\}$)(p'))))
= send(t', q', p')

$\{3\}$ arbitrary(status(q'))

Applying unact_ax and instantiating the top quantifier with the term: $\text{send}(t', q', p')$ and hiding some formulas,

`Validity.2.1:`

$$\{-1\} \quad \text{UnR}(R(\text{send}(t', q', p'))) = \text{send}(t', q', p')$$

$$\{-2\} \quad \text{HybridMajority}(\text{caucus}' - \{q'\}, (\lambda\ (q : \text{fcu}) : \text{OMH}(q, r', R(\text{send}(t', q', q)), \text{caucus}' - \{q'\})(p'))) = R(\text{send}(t', q', p'))$$

$$\{1\} \quad \text{UnR}(\text{HybridMajority}(\text{caucus}' - \{q'\}, (\lambda\ (q : \text{fcu}) : \text{OMH}(q, r', R(\text{send}(t', q', q)), \text{caucus}' - \{q'\})(p')))) = \text{send}(t', q', p')$$

Invoking decision procedures completes the proof of `Validity.2.1`.

⋮

The remainder of the proof takes up over one hundred pages of printed text, and is omitted here. Full machine-readable PVS specifications and PVS proofs of the entire proof chain are available from the authors.

4.3 PVS Proof Chain Analysis

Here we reproduce a summary of the PVS analysis of the entire chain of proof for the verification conducted. Following the summary is a detailed description of all definitions, axioms, assumptions, lemmas, and theorems used implicitly or explicitly in three example proofs.

```
Proof summary for theory omh
    IMPORTING1_TCC1.......................................proved - complete
    fincard_all...........................................proved - complete
    send5.................................................proved - complete
    HybridMajority_TCC1...................................proved - complete
    HybridMajority_ax1....................................proved - complete
    HybridMajority_ax2....................................proved - complete
    OMH_TCC1..............................................proved - complete
    OMH_TCC2..............................................proved - complete
```

```
Validity_Prop_TCC1..................................proved - complete
Validity............................................proved - complete
Agreement_Prop_TCC1.................................proved - complete
Agreement...........................................proved - complete
Validity_Final......................................proved - complete
Validity_Final_TCC1.................................proved - complete
Validity_Final_TCC2.................................proved - complete
Validity_Corollary..................................proved - complete
Validity_Corollary_TCC1.............................proved - complete
Validity_Corollary_TCC2.............................proved - complete
Agreement_Final.....................................proved - complete
Crash_Only_Validity.................................proved - complete
Crash_Only_Agreement................................proved - complete
Crash_Only_Validity_Final...........................proved - complete
Crash_Only_Validity_Final_TCC1......................proved - complete
Crash_Only_Validity_Final_TCC2......................proved - complete
Crash_Only_Validity_Corollary.......................proved - complete
Crash_Only_Agreement_Final..........................proved - complete
Theory totals: 26 formulas, 26 attempted, 26 succeeded.
```

The first example analyzed in detail is *send5*, which depends only on the axiomatization of `send` and `status`.

```
send5 has been PROVED.

The proof chain for send5 is COMPLETE.

send5 depends on the following axioms:
   omh.statuses_inclusive
   omh.send4
   omh.send2
   omh.send1

send5 depends on the following definitions:
   omh.s
   omh.c
   omh.g
   omh.a
```

The second detailed analysis presented here is of the final validity theorem corresponding to Lemma 1 in the semiformal proof described earlier. This theorem is essentially proved by appeal to the inductive version of validity, which has a long and complicated proof whose beginning was presented in Section 4.2.1. By transitivity of dependencies, this final version of validity depends on all the definitions and axioms that the inductive version of validity depends on, plus a few more. Note that this

proof also depends on several lemmas and definitions from the PVS prelude, such as the definitions of *fincard* and *filter*. Lemmas and definitions from the prelude are cited in these proof chain analyses, but the axioms of propositional logic, equality, the lambda calculus, and linear arithmetic used implicitly by the ground decision procedures are not identified in this way.

```
Validity_Final has been PROVED.

The proof chain for Validity_Final is COMPLETE.

Validity_Final depends on the following proved theorems:
   bounded_induction.upto_induction
   omh.Validity
   finite_cardinality.fincard_TCC1
   hybridmjrty.Hybrid_mjrty_TCC1
   omh.send5
   hybridmjrty.Hybrid_mjrty_TCC3
   omh.OMH_TCC1
   finite_cardinality.fincardi_TCC3
   card_set.fullset_fincard
   hybridmjrty.count_votes_TCC4
   finite_cardinality.fincardi_TCC1
   hybridmjrty.count_votes_TCC3
   hybridmjrty.Hinv_holds
   omh.OMH_TCC2
   hybridmjrty.Hybrid_mjrty_TCC2
   omh.Validity_Final_TCC2
   hybridmjrty.Hlosers
   hybridmjrty.Hwinner
   omh.IMPORTING1_TCC1
   hybridmjrty.count_votes_TCC5
   omh.HybridMajority_ax1
   hybridmjrty.count_all_good_votes_TCC2
   identity.I_TCC2
   hybridmjrty.Hybrid_mjrty_TCC5
   hybridmjrty.count_votes_TCC2
   hybridmjrty.count_all_good_votes_TCC3
   omh.Validity_Prop_TCC1
   card_set.remove_comm
   hybridmjrty.count_votes_TCC1
   finite_cardinality.fincardi_TCC2
   card_set.remove_prop
   omh.Validity_Final_TCC1
   card_set.fincard_remove
   hybridmjrty.count_all_good_votes_TCC1
   omh.fincard_all
```

```
   finite_cardinality.fincardi_TCC4
   omh.HybridMajority_TCC1
   card_set.fincard_filter
   hybridmjrty.count_all_good_votes_TCC4

Validity_Final depends on the following axioms:
   omh.send2
   omh.statuses_disjoint
   omh.send1
   omh.statuses_inclusive
   omh.send4

Validity_Final depends on the following definitions:
   identity.I
   omh.c
   finite_cardinality.fincard
   sets.fullset
   omh.gs
   filters.filter
   omh.a
   omh.Validity_Prop
   identity.identity
   hybridmjrty.Hinv
   omh.cs
   hybridmjrty.Hybrid_mjrty
   omh.HybridMajority
   sets.member
   finite_cardinality.fincardi
   hybridmjrty.count_votes
   omh.g
   omh.as
   hybridmjrty.count_all_good_votes
   omh.OMH
   omh.ss
   omh.s
   sets.remove

Validity_Final depends on the following assumptions:
   omh.unact_ax
   omh.act_ax
```

The third detailed analysis is that of the agreement property. The proof of this theorem depends on the inductive version of agreement, which in turn depends on the inductive validity property.

```
Agreement_Final has been PROVED.

The proof chain for Agreement_Final is COMPLETE.

Agreement_Final depends on the following proved theorems:
   bounded_induction.upto_induction
   omh.Validity
   finite_cardinality.fincard_TCC1
   hybridmjrty.Hybrid_mjrty_TCC1
   omh.send5
   hybridmjrty.Hybrid_mjrty_TCC3
   omh.OMH_TCC1
   finite_cardinality.fincardi_TCC3
   card_set.fullset_fincard
   hybridmjrty.count_votes_TCC4
   finite_cardinality.fincardi_TCC1
   hybridmjrty.count_votes_TCC3
   hybridmjrty.count_votes_TCC1
   omh.OMH_TCC2
   hybridmjrty.Hybrid_mjrty_TCC2
   hybridmjrty.Hlosers
   omh.Validity_Corollary_TCC1
   hybridmjrty.Hwinner
   omh.IMPORTING1_TCC1
   hybridmjrty.count_votes_TCC5
   omh.HybridMajority_ax1
   hybridmjrty.count_all_good_votes_TCC2
   omh.Agreement
   identity.I_TCC2
   omh.Agreement_Prop_TCC1
   hybridmjrty.count_votes_TCC2
   hybridmjrty.count_all_good_votes_TCC3
   omh.Validity_Prop_TCC1
   card_set.remove_comm
   hybridmjrty.Hybrid_mjrty_TCC5
   hybridmjrty.Hinv_holds
   omh.HybridMajority_ax2
   card_set.fincard_non_empty
   omh.Validity_Corollary_TCC2
   finite_cardinality.fincardi_TCC2
   card_set.remove_prop
   card_set.fincard_remove
```

```
  hybridmjrty.count_all_good_votes_TCC1
  omh.fincard_all
  finite_cardinality.fincardi_TCC4
  omh.HybridMajority_TCC1
  card_set.fincard_filter
  hybridmjrty.count_all_good_votes_TCC4

Agreement_Final depends on the following axioms:
  omh.send2
  omh.statuses_disjoint
  omh.send1
  omh.statuses_inclusive
  omh.send4

Agreement_Final depends on the following definitions:
  identity.I
  omh.c
  finite_cardinality.fincard
  sets.fullset
  omh.gs
  omh.a
  omh.Validity_Prop
  identity.identity
  hybridmjrty.Hinv
  omh.cs
  hybridmjrty.Hybrid_mjrty
  omh.HybridMajority
  sets.member
  finite_cardinality.fincardi
  hybridmjrty.count_votes
  omh.g
  omh.Agreement_Prop
  omh.as
  filters.filter
  hybridmjrty.count_all_good_votes
  omh.OMH
  omh.ss
  omh.s
  sets.remove

Agreement_Final depends on the following assumptions:
  omh.unact_ax
  omh.act_ax
```

The complete detailed proof chain analysis of every lemma from the *omh* theory is over 20 pages in length.

Chapter 5

Conclusions

Tools for formal verification have matured to the point where complex, practically interesting aspects of systems can be economically verified. The human effort required to specify and prove in complete formal detail interesting theorems about fault-tolerant architectures is quite modest. In this report we have presented the formal verification of a new algorithm for Byzantine Agreement under a hybrid fault model.

Thambidurai and Park's hybrid fault model extends the design and analysis of Byzantine fault-tolerant algorithms in an important and useful way. Hybrid fault-tolerant algorithms can tolerate greater numbers of "simple" faults than classical Byzantine fault-tolerant algorithms, without sacrificing the ability to withstand Byzantine, or arbitrary, faults. We applied our formal verification tools to this domain, discovering errors in published proofs and in a proposed algorithm for Byzantine Agreement under this fault model.

A crucial tool in our detection of the flaw in Thambidurai and Park's algorithm, and also in detecting flaws in our own early attempts to repair this algorithm, was our use of mechanically-checked formal verification. The discipline of formal specification and verification was also instrumental in helping us to develop the correct algorithm presented here. The rigor of a mechanically-checked proof enhances our conviction that this algorithm is, indeed, correct, and also helped us develop the informal, but detailed, proof given here in the style of a traditional mathematical presentation.

It is worth repeating that no formal verification proves any program "correct." At most, a model of the program is shown to satisfy a specification, and shown to exhibit certain properties under a certain set of assumptions. The true benefit of formal specification and verification is *not* in getting a theorem prover to say **proved**, but rather in refining one's understanding through dialogue with a tireless mechanical skeptic.

The effort required to perform this formal verification was not particularly large and did not seem to us to demand special skill. We attribute some of this ease in performing formal verification of a relatively tricky algorithm to the effectiveness of the tools employed [22]. These tools (and others that may be of similar effectiveness) are freely available. In light of the flaws we discovered in Thambidurai and Park's algorithm, and had previously found in the proofs for other fault-tolerant algorithms [24, 29, 30], we suggest that formal verification should become a routine part of the social process of development and analysis of fault-tolerant algorithms intended for practical application in safety-critical systems.

In future work, we hope to explore possible extensions to the OMH algorithm and its analysis to include communication faults, and to see whether larger numbers of symmetric faults can be tolerated. We also intend to study whether lower message complexity can be achieved in cases of practical interest, and to examine alternative architectures employing fewer processors (we have already formally specified and verified a variant of OMH(1) for the asymmetric Draper FTP architecture [13]). We also plan to formally verify a modified version of the Interactive-Convergence Algorithm for clock synchronization using a hybrid fault model that includes communication faults (we have already formally verified the standard algorithm [28], and have an informal analysis of the modified version).

Also, although our experience indicates that formal verification is an effective debugging technique, it is undeniably expensive one, and it is interesting to ask whether other methods could have identified the flaws in Z and its derivatives more simply or economically. Our previous experience with other mechanically-checked verifications is consistent with the experience reported here: the effort spent discovering and repairing flaws in a specification, algorithm, or proof is a large part of the intellectual effort expended on formal verification projects.

An obvious alternative is testing: our specifications of these algorithms can be easily translated into Lisp or other higher-order functional languages where they can be run on a variety of test cases. (Obviously, controlled "fault-injections" will need to be programmed into the executable algorithms.) Without specifying a particular strategy for determining test cases, we cannot say whether specific flaws could have been detected in this way or not. However, it is clear that with less than complete test coverage, one cannot guarantee that all errors will be discovered.

Between testing and conventional verification lie the *state-exploration* methods. These methods resemble testing in that they are automatic; they resemble verification in that they *are* formal verification methods. State-exploration methods systematically enumerate all the states of a finite-state algorithm and test whether certain predicates hold at those states. Recent techniques allow large numbers of

states to be handled in an efficient manner.[1] Several systems based on state exploration are available; some of these exploit the close connection between finite state graphs and propositional temporal logic (when they are usually called "model checkers" [8]), others provide a higher-level language (e.g., Murϕ [10, 19] uses a transition-rule language for concurrent systems that is loosely based on Chandy and Misra's Unity model [7]).

As it stands, OMH is not amenable to state exploration: it has far too many states. But for debugging, it could be sufficient to examine highly simplified versions of the problem: for example, the case $m = 1$, $n \leq 6$, and a very small set of data values—E, $R(E)$, and three distinct "good" values—seem sufficient to detect all the errors that we discovered.

Whereas conventional testing probes selected test cases of the full algorithm, state exploration provides complete coverage of simplified instances. We plan to examine the effectiveness of state exploration in this domain by conducting some experiments with OMH and related algorithms.

[1] These techniques include hashing [12], and symbolic methods using Binary Decision Diagrams [5, 6].

Bibliography

[1] William R. Bevier and William D. Young. Machine-checked proofs of the design and implementation of a fault-tolerant circuit. NASA contractor report 182099, NASA Langley Research Center, Hampton, VA, November 1990. (Work performed by Computational Logic Incorporated).

[2] William R. Bevier and William D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4(6A):755–775, 1992.

[3] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[4] M. Broy. Experience with software specification and verification using LP, the Larch Proof assistant. Technical Report 93, DEC Systems Research Center, Palo Alto, CA, November 1992.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[9] D. Davies and J. Wakerley. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[11] Danny Dolev, Michael J. Fisher, Rob Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for Byzantine Agreement without authentication. *Information and Control*, 52:257–274, 1982.

[12] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[13] Albert L. Hopkins, Jr., Jaynarayan H. Lala, and T. Basil Smith III. The evolution of fault tolerant computing at the Charles Stark Draper Laboratory, 1955–85. In A. Avižienis, H. Kopetz, and J. C. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1 of *Dependable Computing and Fault-Tolerant Systems*, pages 121–140. Springer Verlag, Wien, Austria, 1987.

[14] M. McElvany Hugue. Estimating reliability under the static hybrid fault model. Technical report, Aerospace Technology Center, Allied-Signal Aerospace Company, Columbia, MD, 1992.

[15] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[16] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[17] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[18] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[19] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual.* Computer Science Department, Stanford University, Stanford, CA, March 1993.

[20] Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. In *Fault Tolerant Computing Symposium 17*, pages 48–54, Pittsburgh, PA, July 1987. IEEE Computer Society.

[21] Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, April 1991.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer Verlag.

[23] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[24] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Some lessons learned. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 482–500, Odense, Denmark, April 1993. Springer Verlag.

[25] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[26] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–481, March 1986.

[27] John Rushby. Formal verification of an Oral Messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992.

[28] John Rushby and Friedrich von Henke. Formal verification of the Interactive Convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Original version also available as NASA Contractor Report 4239, June 1989.

[29] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[30] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes*

in Computer Science, pages 217–236, Nijmegen, The Netherlands, January 1992. Springer Verlag.

[31] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[32] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[33] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.

[34] Philip Thambidurai, You-Keun Park, and Kishor S. Trivedi. On reliability modeling of fault-tolerant distributed systems. In *9th International Conference on Distributed Computing Systems*, pages 136–142, Newport Beach, CA, June 1989. IEEE Computer Society.

[35] Chris J. Walter. Identifying the cause of detected errors. In *Fault Tolerant Computing Symposium 20*, pages 48–55, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[36] John H. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[37] William D. Young. Verifying the Interactive Convergence clock-synchronization algorithm using the Boyer-Moore prover. NASA Contractor Report 189649, NASA Langley Research Center, Hampton, VA, April 1992. (Work performed by Computational Logic Incorporated).

Appendix A

Hybrid MJRTY

In the proofs performed to verify the correctness of OMH, only two properties about *HybridMajority* are assumed. Any implementation of *HybridMajority* that satisfies these axioms would be acceptable for the purposes of OMH.

$$
\begin{aligned}
&|\mathrm{gs}(\mathrm{caucus})| \; > \; |\mathrm{as}(\mathrm{caucus})| + |\mathrm{ss}(\mathrm{caucus})| \\
&\quad \wedge \; (\forall\, p : \; g(p) \wedge p \in \mathrm{caucus} \supset v(p) = t) \\
&\quad\quad \wedge \; t \neq \mathrm{error} \; \wedge \; (\forall\, p : \; c(p) \wedge p \in \mathrm{caucus} \supset v(p) = \mathrm{error}) \\
&\quad \supset \mathrm{HybridMajority}(\mathrm{caucus}, v) = t
\end{aligned}
$$

$$
\begin{aligned}
&(\forall\, p : \; p \in \mathrm{caucus} \supset v_1(p) = v_2(p)) \\
&\quad \supset \mathrm{HybridMajority}(\mathrm{caucus}, v_1) = \mathrm{HybridMajority}(\mathrm{caucus}, v_2)
\end{aligned}
$$

These properties were described and justified in detail in Section 4.1. Nonetheless, there is always a concern when properties are stated axiomatically that they might be unrealizable. We allay this concern by exhibiting an efficient implementation for the *HybridMajority* function and proving that it satisfies the stated axioms.

This development is specified in the PVS theory ***hybridmjrty*** (shown on page 62), which constructively specifies a function ***Hybrid_mjrty***. and then shows that it can be used to satisfy the axioms stated above. The function ***Hybrid_mjrty*** is very similar to the Boyer-Moore MJRTY algorithm [3], except that it ignores error values. MJRTY is a method for finding the absolute majority (if there is one) of a set of values in linear time, using only equality comparison operations. (Other implementations of a majority function require more than linear time, and/or more complex comparisons).

The original MJRTY algorithm has been formally specified and verified before [3,4]. The modified version below is based on a recent unpublished verification of MJRTY by Natarajan Shankar.

MJRTY can be informally explained as a method to find the candidate, if there is one, with the majority of votes at a political convention. Every voter on the convention floor attempts to find someone who is voting for some other candidate. If two voters voting for different candidates meet, they annihilate each other. At the end of this process, all the remaining voters must support the same candidate. The key property of this procedure is that if there is a majority for some candidate in the beginning, then there will be some voters for that candidate left at the end. If there is no candidate with a majority, then there may or may not be any voters left at the end, and those left could be voting for any candidate, even the one with the fewest total votes. Thus, a second (linear) pass is necessary to ensure that the delegate supported by the voters remaining at the end does indeed have an overall majority.

This violently parallel procedure can be sequentialized by ordering the voters, and then moving down the line of voters forming a "bandwagon" of like-minded voters. When this bandwagon finds that the next voter agrees with them, the bandwagon simply increases in size by one. When it finds that the next voter choses some other candidate, the bandwagon's size is decreased by one. If the bandwagon becomes empty, then the next voter becomes a bandwagon of one, and the process continues. This procedure can be implemented by storing the candidate chosen by the current bandwagon, and a natural number representing the size of the bandwagon.

It is straightforward to generalize this procedure to a version which respects caucuses and ignores errors, as required by OMH. The input to the generalized procedure is a set of votes possibly containing votes belonging to processors outside the caucus, or in error, which should both be ignored in selecting the majority. It would be possible to use the standard MJRTY function on a set of votes filtered to remove error and noncaucus votes, although such a specification would be somewhat farther from the most efficient implementation, requiring additional passes over the set of votes. The generalized procedure must also specify the value to return in the case that there are no nonerror votes in the caucus.

The sequential algorithm is represented in our specification as the function *Hybrid_mjrty*, which takes a *caucus* (a set of eligible voters), a *poll* (a function from all voters to their chosen candidate), and i (the total number of votes) as arguments and returns a pair consisting of the identity of a candidate, and a natural number standing for the size of the bandwagon of voters remaining at the end. In the base case, the default value is chosen to be the error value. The function *count_votes* counts the votes of a particular candidate, while *count_all_good_votes* counts the total number of nonerror votes in a caucus.

Hlosers is a lemma about *Hybrid_mjrty* stating that for all candidates A, if A is not the first element of the pair returned by *Hybrid_mjrty*, then A does not have a majority of good votes. *Hwinner* is closely related to *Hlosers*, stating that if a candidate has a majority of votes, then that candidate will be selected by *Hybrid_mjrty*.

The properties *Hlosers* and *Hwinner* follow directly from the inductive invariant *Hinv*. The property *Hinv* compactly asserts two invariants at once. First, that twice the total of the selected candidates votes is less than the size of the current bandwagon plus the total number of good (nonerror, in caucus) votes. Second, that twice the total of the selected candidates votes plus the size of the current bandwagon is less than the total number of good votes. These invariants are true for all nonerror candidates. *Hinv* is proven by induction on the total number of votes.

```
hybridmjrty[T : TYPE, n : posnat, error : T] : THEORY
  BEGIN
  poll : VAR [below[n] → T]
  caucus : VAR setof[below[n]]
  i : VAR upto[n]
  A, cand : VAR T
  P : VAR [T, below[n]]

  Hybrid_mjrty(caucus, poll, i) : RECURSIVE [T, nat] =
    (IF i = 0 THEN (error, 0)
     ELSE
      (LET P = Hybrid_mjrty(caucus, poll, i − 1) IN
        (IF (¬ caucus(i − 1)) ∨ poll(i − 1) = error THEN P
         ELSIF poll(i − 1) = proj_1(P) THEN (proj_1(P), proj_2(P) + 1)
         ELSIF proj_2(P) > 0 THEN (proj_1(P), proj_2(P) − 1)
         ELSE (poll(i − 1), 1)
         ENDIF))
     ENDIF)
   MEASURE (λ caucus, poll, i : i)

  count_votes(caucus, poll, cand, i) : RECURSIVE nat =
    (IF i > 0
      THEN
      (IF caucus(i − 1) ∧ poll(i − 1) = cand THEN 1
           + count_votes(caucus, poll, cand, i − 1)
       ELSE count_votes(caucus, poll, cand, i − 1)
       ENDIF)
     ELSE 0
```

```
    ENDIF)
  MEASURE (λ caucus, poll, cand, i : i)

 count_all_good_votes(caucus, poll, i) : RECURSIVE nat =
   (IF i > 0
      THEN
     (IF caucus(i - 1) ∧ (¬ poll(i - 1) = error)
        THEN 1 + count_all_good_votes(caucus, poll, i - 1)
      ELSE count_all_good_votes(caucus, poll, i - 1)
      ENDIF)
    ELSE 0
    ENDIF)
  MEASURE (λ caucus, poll, i : i)

 Hinv(caucus, poll, i) :
  bool
    =
    (LET P = Hybrid_mjrty(caucus, poll, i) IN
      (∀ A :
        (¬ A = error)
          ⊃ 2
           ×
          (count_votes(caucus, poll, A, i)
             + (IF A = proj_1(P) THEN 0 ELSE proj_2(P) ENDIF))
           ≤ proj_2(P) + count_all_good_votes(caucus, poll, i)))

 Hinv_holds : LEMMA Hinv(caucus, poll, i)

 Hlosers :
  LEMMA
   A ≠ proj_1(Hybrid_mjrty(caucus, poll, i)) ∧ (¬ A = error)
     ⊃ 2 × count_votes(caucus, poll, A, i)
      ≤ count_all_good_votes(caucus, poll, i)

 Hwinner :
  LEMMA
   (∀ cand :
     (2 × count_votes(caucus, poll, cand, i)
         > count_all_good_votes(caucus, poll, i)
         ∧ ¬ cand = error)
        ⊃ proj_1(Hybrid_mjrty(caucus, poll, i)) = cand)

 END hybridmjrty
```

The proof of TCC's and lemmas from the *hybridmjrty* theory have been completed in PVS. Informally, there is only one significant lemma, *Hinv_holds*, which is proved by induction on i. In the base case, this holds by definition of the functions. In the inductive case, there are many cases, but each one is relatively straightforward to analyze. The remaining two lemmas are immediately provable from *Hinv_holds*. The following is the proof chain analysis from PVS.

```
Proof summary for theory hybridmjrty
    Hybrid_mjrty_TCC5....................................proved - complete
    Hybrid_mjrty_TCC1....................................proved - complete
    Hybrid_mjrty_TCC2....................................proved - complete
    Hybrid_mjrty_TCC3....................................proved - complete
    count_votes_TCC1.....................................proved - complete
    count_votes_TCC2.....................................proved - complete
    count_votes_TCC3.....................................proved - complete
    count_votes_TCC4.....................................proved - complete
    count_votes_TCC5.....................................proved - complete
    count_all_good_votes_TCC1............................proved - complete
    count_all_good_votes_TCC2............................proved - complete
    count_all_good_votes_TCC3............................proved - complete
    count_all_good_votes_TCC4............................proved - complete
    Hinv_holds...........................................proved - complete
    Hlosers..............................................proved - complete
    Hwinner..............................................proved - complete
    Theory totals: 16 formulas, 16 attempted, 16 succeeded.


Hinv_holds has been PROVED.

The proof chain for Hinv_holds is COMPLETE.

Hinv_holds depends on the following proved theorems:
   hybridmjrty.Hybrid_mjrty_TCC3
   hybridmjrty.count_all_good_votes_TCC1
   hybridmjrty.count_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC2
   hybridmjrty.count_all_good_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC3
   hybridmjrty.count_votes_TCC5
   hybridmjrty.Hybrid_mjrty_TCC5
   bounded_induction.upto_induction
   hybridmjrty.count_votes_TCC3
   hybridmjrty.count_votes_TCC1
   hybridmjrty.Hybrid_mjrty_TCC1
   hybridmjrty.count_votes_TCC2
   hybridmjrty.Hybrid_mjrty_TCC2
```

```
Hinv_holds depends on the following definitions:
   hybridmjrty.count_votes
   hybridmjrty.Hybrid_mjrty
   hybridmjrty.count_all_good_votes
   hybridmjrty.Hinv


Hlosers has been PROVED.

The proof chain for Hlosers is COMPLETE.

Hlosers depends on the following proved theorems:
   hybridmjrty.Hybrid_mjrty_TCC3
   hybridmjrty.count_all_good_votes_TCC1
   hybridmjrty.count_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC2
   hybridmjrty.count_all_good_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC3
   hybridmjrty.count_votes_TCC5
   hybridmjrty.Hybrid_mjrty_TCC5
   bounded_induction.upto_induction
   hybridmjrty.count_votes_TCC3
   hybridmjrty.Hinv_holds
   hybridmjrty.count_votes_TCC1
   hybridmjrty.Hybrid_mjrty_TCC1
   hybridmjrty.count_votes_TCC2
   hybridmjrty.Hybrid_mjrty_TCC2

Hlosers depends on the following definitions:
   hybridmjrty.count_votes
   hybridmjrty.Hybrid_mjrty
   hybridmjrty.count_all_good_votes
   hybridmjrty.Hinv


Hwinner has been PROVED.

The proof chain for Hwinner is COMPLETE.

Hwinner depends on the following proved theorems:
   hybridmjrty.Hybrid_mjrty_TCC3
   hybridmjrty.count_all_good_votes_TCC1
   hybridmjrty.Hlosers
   hybridmjrty.count_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC2
```

```
   hybridmjrty.count_all_good_votes_TCC4
   hybridmjrty.count_all_good_votes_TCC3
   hybridmjrty.count_votes_TCC5
   hybridmjrty.Hybrid_mjrty_TCC5
   bounded_induction.upto_induction
   hybridmjrty.count_votes_TCC3
   hybridmjrty.Hinv_holds
   hybridmjrty.count_votes_TCC1
   hybridmjrty.Hybrid_mjrty_TCC1
   hybridmjrty.count_votes_TCC2
   hybridmjrty.Hybrid_mjrty_TCC2

Hwinner depends on the following definitions:
   hybridmjrty.count_votes
   hybridmjrty.Hybrid_mjrty
   hybridmjrty.count_all_good_votes
   hybridmjrty.Hinv
```

A.1 Using Hybrid_mjrty

Given *hybrid_mjrty*, the following definition is sufficient to satisfy the axioms for *HybridMajority* stated earlier in the theory omh:

$$\text{HybridMajority}(\text{caucus}, v) : T \ = \ \text{proj_1}(\text{Hybrid_mjrty}(\text{caucus}, v, n))$$

The proof of satisfaction of the second axiom given for *Hybridmajority* from *Hlosers* and *Hwinner* is a relatively straightforward induction on the number of votes. The proof of satisfaction of the first axiom from *Hwinner* is much more intricate. The proof proceeds by effectively introducing two lemmas from which the proof of the desired property is relatively straightforward. The lemmas are introduced by the PVS proof command **case**, which splits the proof into two branches: on one branch the lemma can be assumed, on the other it must be proved. The first lemma states that if all good processors in a caucus agree on a candidate (value), then *count_votes* returns a value at least as big as the number of good processors.

$$\begin{array}{l}(\forall\ (p :\ \text{fcu}) :\ g(p)\ \wedge p \in \text{caucus}\ \supset\ v(p) = t) \\ \quad \supset\ |gs(\text{caucus})| \leq \text{count_votes}(\text{caucus}, v, t, n)\end{array}$$

The other lemma states that if all good processors vote for a nonerror value, and all manifest-faulty processors vote for an error value, then the sum of the cardinalities of good, arbitrary, and symmetric faulty processors is at least as large as the value returned by *count_all_good_votes*.

$$
\begin{array}{l}
(\forall\ (p:\ \text{fcu}):\ g(p)\ \wedge\ p \in \text{caucus}\ \supset\ v(p) = t) \\
\wedge\ (\forall\ (p:\ \text{fcu}):\ c(p)\ \wedge\ p \in \text{caucus}\ \supset\ v(p) = \text{error}) \\
\supset \\
\quad |gs(\text{caucus})| + |as(\text{caucus})| + |ss(\text{caucus})| \geq \text{count_all_good_votes}(\text{caucus}, v, n) \\
\qquad \vee\ t = \text{error}
\end{array}
$$

Both lemmas are proven by induction on the number of votes. These lemmas do not appear in the formal specification, since they are only used internally in proving the property *Hybridmajority_ax1*. In fact with subtle proof manipulation the use of these lemmas could be eliminated from the proof, although doing so would lengthen the proof. From these lemmas and *Hwinner* the property *Hybridmajority_ax1* follows directly.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE July 1993	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE
A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model

5. FUNDING NUMBERS
C NAS1-18969
WU 505-64-10-13

6. AUTHOR(S)
Patrick Lincoln and John Rushby

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-4527

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Rick W. Butler
Final Report - Task 6

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Consistent distribution of single-source data to replicated computing channels is a fundamental problem in fault-tolerant system design. The "Oral Messages" (OM) algorithm solves this problem of Interactive Consistency (Byzantine Agreement) assuming that all faults are worst-case. Thambidurai and Park introduced a "hybrid" fault model that distinguished three fault modes: asymmetric (Byzantine), symmetric, and benign; they also exhibited, along with an informal "proof of correctness," a modified version of OM. Unfortunately, their algorithm is flawed. The discipline of mechanically checked formal verification eventually enabled us to develop a correct algorithm for Interactive Consistency under the hybrid fault model. This algorithm withstands a asymmetric, s symmetric, and b benign faults simultaneously, using $m+1$ rounds, provided $n> 2a + 2s + b + m$, and $m\geq a$. We present this algorithm, discuss its subtle points, and describe its formal specification and verification in PVS. We argue that formal verification systems such as PVS are now sufficiently effective that their application to fault-tolerance algorithms should be considered routine.

14. SUBJECT TERMS
Byzantine agreement, Hybrid fault model, Interactive consistency, Formal verification

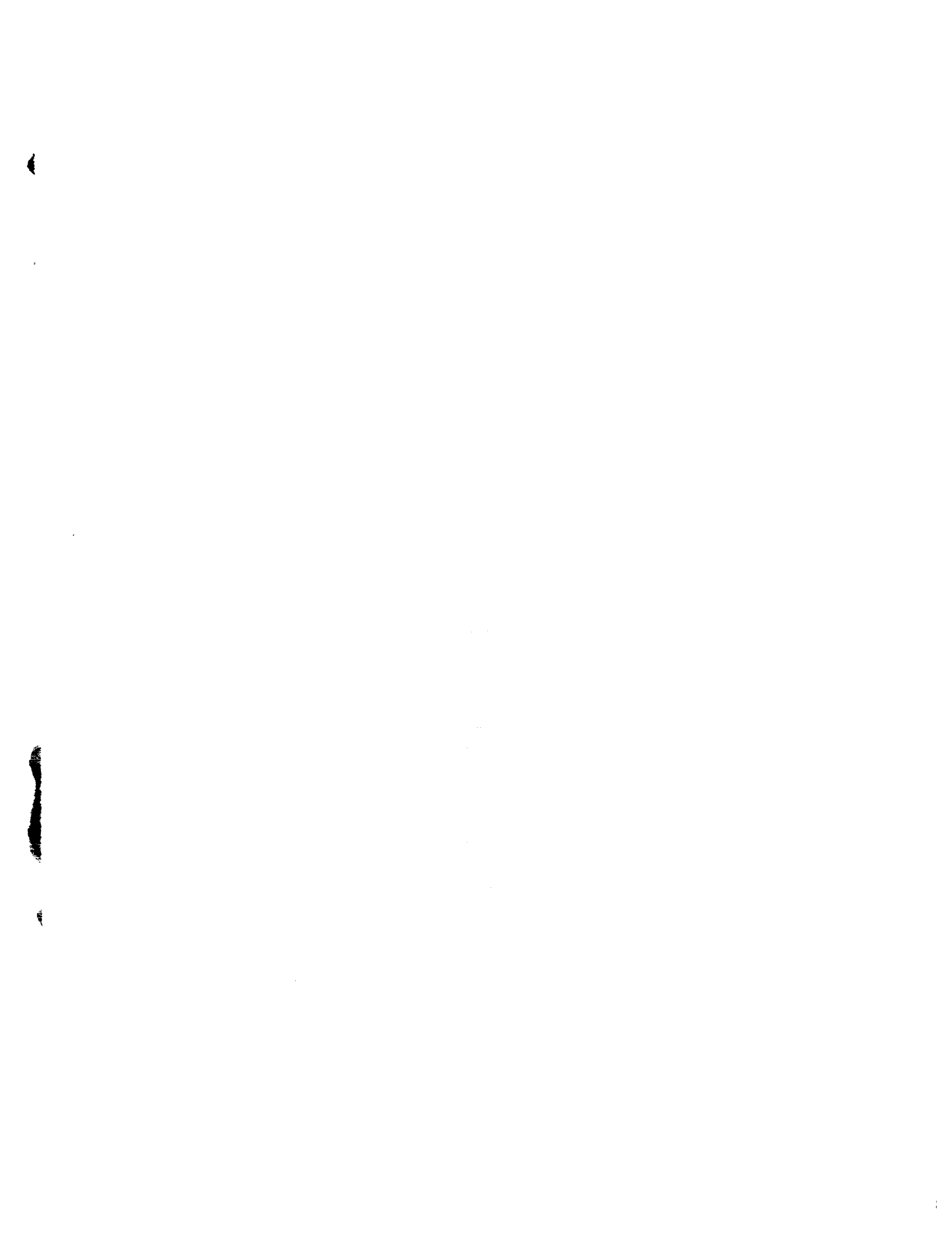
15. NUMBER OF PAGES
76

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102



BULK RATE
POSTAGE & FEES PAID
NASA
Permit No. G-27

POSTMASTER: If Undeliverable (Section 158 Postal Manual) Do Not Return