The Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model*

Preprint of a paper to be presented at the Conference on Computer-Aided
Verification, CAV '93, Elounda, Crete, Greece, June-July 1993

Patrick Lincoln and John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Lincoln@csl.sri.com Rushby@csl.sri.com
Phone: +1 (415) 859-5454 Fax: +1 (415) 859-2844

Abstract

Modern verification systems such as PVS are now reaching the stage of development where the formal verification of critical algorithms is feasible with reasonable effort. This paper describes one such verification in the field of fault tolerance. The distribution of single-source data to replicated computing channels (Interactive Consistency or Byzantine Agreement) is a central problem in this field. The classic Oral Messages (OM) algorithm solves this problem under the assumption that all channels are either nonfaulty or arbitrarily (Byzantine) faulty. Thambidurai and Park have introduced a "hybrid" fault model that distinguishes additional fault modes, along with a modified version of OM. They gave an informal proof that their algorithm withstands the same number of arbitrary faults, but more "nonmalicious" faults than OM.

We detected a flaw in this algorithm while undertaking its formal verification using PVS. The discipline of mechanically-checked formal verification helped us to develop a corrected version of the algorithm. Here we describe the formal specification and verification of this new algorithm. We argue that formal verification systems such as PVS are now sufficiently effective that their application to critical fault-tolerance algorithms should be considered routine.

*This work was sponsored by the National Aeronautics and Space Administration Langley Research Center under contract NAS1 18969.

1 Introduction

Improved verification systems, new techniques, and modern workstations allow formal verification of interesting properties of critical or complex systems to be undertaken with reasonable effort. For maximum utility, formal verification should focus on those aspects of design and analysis that are least well served by conventional techniques, and where errors can be catastrophic or expensive. Because there are examples of safety-critical systems where unanticipated behaviors of the mechanisms for fault-tolerance became the ***primary*** source of system failure [7], we view these mechanisms as a fruitful target for formal verification energies. In this paper we describe the formal verification of a difficult algorithm used in fault-tolerant architectures, and illustrate some of the characteristics of a recently developed verification system, called PVS [8, 9], with which we undertook the verification. The theorems proved in this paper are not especially deep, but they do require extreme attention to detail (involving many case splits and large numbers of linear inequalities that vary subtly from case to case) and seem prone to error in the social process of human proof checking. The interactive theorem prover of PVS provides rather powerful automation, including decision procedures for arithmetic, and allows the user to check these arguments without becoming lost in detail.

In this paper, we focus on algorithms for distributing single-source data to multiple channels in the presence of faults. This problem, variously known as "interactive consistency," "Byzantine Agreement," or "source congruence" was first posed and solved in 1980 [4, 10] with the "Oral Messages" algorithm (abbreviated to OM). (This algorithm has been formally specified and verified by Bevier and Young [1] and, subsequently, by us [11].) Algorithm OM makes no assumptions about the possible kinds of faults. An alternative is to develop a detailed model of the kinds of faults expected and to construct an algorithm tuned to deliver maximum resilence to those particular fault "modes." However, an overlooked fault mode may then cause unexpected failure in system operation. Thus arise ***hybrid*** fault models that include the arbitrary, or Byzantine fault mode, together with a limited number of additional fault modes. Inclusion of the arbitrary fault mode eliminates the fear that some unforeseen behavior may defeat the fault-tolerance mechanisms, while inclusion of other fault modes allows greater resilence to faults of these particular (and common) kinds than a classical Byzantine fault-tolerant architecture.

Thambidurai and Park [14] introduced Algorithm Z, a variant of OM that is claimed to retain the effectiveness of OM with respect to arbitrary faults, but that is also capable of withstanding more faults of the other kinds considered than is OM. We attempted to formally verify the published proof of correctness for Algorithm Z, but failed. Eventually, our investigation led us to discover circumstances under which Algorithm Z fails. We developed several patches for the algorithm, along with rather convincing informal proofs of their correctness. However, in attempting formal verification of these proofs we found flaws in them, too. We finally produced an algorithm (which we call OMH, for Oral Messages Hybrid) and a set of informal proofs that we were able to formally verify using the PVS verification system. Detailed discussion of the algorithm, and of its flawed variants, is the topic of another paper [6]; here, we outline the formal verification and show that once the correct algorithm is obtained, the proofs are not particularly hard. Because informal proofs are demonstrably unreliable in this domain, and because the consequences of failure could be catastrophic, we argue that formal verification should become routine.

2 The Problem: Assumptions and Requirements

The goal is to distribute single-source data (such as a sensor sample) from one processor of a fault-tolerant system to all the others in such a way that all nonfaulty processors receive the same value. The principal difficulty is that a faulty processor may send different values to different receivers. To overcome this, OM-like 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, we make the following assumptions.

There are n processors in total, one of which is distinguished as the transmitter (or the General, in more informal terms). Every message that is sent between nonfaulty processors is correctly delivered. The absence of a message can be detected. The receiver of a message knows who sent it. The fault modes we distinguish for processors are ***nonfaulty***, ***arbitrary-faulty***, ***symmetric-faulty***, and ***manifestly-faulty***. (These correspond to Thambidurai and Park's nonfaulty, asymmetric malicious, symmetric malicious, and nonmalicious faults, respectively; we find these anthropomorphic terms unhelpful.) We specify these fault modes semi-formally as follows. 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 (a distinguished value) if the transmitter is manifest-faulty [1]; unknown if the transmitter is symmetric-faulty, but all receivers obtain the *same* value; and completely unconstrained if the transmitter is arbitrary-faulty. The intuition is that manifest faults represent likely symptoms of processors that have crashed or lost synchronization, while symmetric faults are likely symptoms of processors with corrupted memory locations. Both these kinds of faults may be expected to be much more common than arbitrary faults, and we

[1] Some preprocessing of timeouts, parity or checksums, etc. may be necessary to identify manifestly faulty values. Note that manifest-faults must be symmetric. If a processor were to "crash" in the middle of a protocol, it would be counted as arbitrary-faulty.

would like to be able to withstand as many of them as possible, for a given level of redundancy.

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 requirements. **Agreement**: If processors p and q are nonfaulty, then they agree on the value ascribed to the transmitter; that is: $\nu_p = \nu_q$. **Validity**: If processor p is nonfaulty, and the transmitter is not arbitrary-faulty, the value ascribed to the transmitter by p is the value actually sent from v to p.

3 The Solution: Algorithm OMH

The basic design of OMH is similar to OM, where the transmitter first sends a value to all other processors. Then each receiver plays the part of the transmitter in a recursive instance of the algorithm. Each receiver then takes a vote of the values it has received and uses the majority value as its final value. OMH differs from OM in 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*." If a manifestly bad value is received, it is recorded as the special value E. After several rounds, one could imagine values corresponding to "I'm reporting that he's reporting that she's reporting E" arise. When taking the majority vote, processors ignore all E values, but treat "I'm reporting E" values as regular values. After the majority vote, if the result is "He is reporting Y," then "Y" is taken as the final value. The algorithm OMH(m) is defined semi-formally below. The parameter m is the number of rounds of message exchanges that are to be performed; the functions R and *UnR* correspond to the addition and removal of the "I'm reporting" tags and are specified in more detail shortly.

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 computes the majority of all non-E 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.

3.1 Informal Proof: Sketch

For this algorithm to be correct as stated, we must make three more assumptions: (1) The class of possible messages exchanged between processors can be increased to accommodate new kinds of messages such as "I'm reporting that she's reporting 5"; (2) For all values v, $R(v) \neq E$ (reported errors are never mistaken for errors); (3) For all values v, ***UnR***$(R(v)) = v$ (untagging a tagged value results in the original value.) The addition of the tagging and untagging functions R and ***UnR*** is what distinguishes Algorithm OMH from its flawed parent, Algorithm Z.

The argument for the correctness of OMH is an adaptation of that for the Byzantine Generals formulation of OM [4, page 390]. We define

- n, the number of processors,
- a, the maximum number of arbitrary-faulty processors
- s, the maximum number of symmetric-faulty processors
- c, the maximum number of manifest-faulty processors
- m the number of rounds of message passing the algorithm is to perform.

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

Proof: This is proved by induction on m. In the base case, the assumptions on message transmission ensure the property. For the inductive case, in step 2 of the algorithm all nonfaulty receivers apply the algorithm OMH($m-1$) to the value received. By a counting argument we apply the inductive hypothesis to conclude that nonfaulty receivers correctly record the forwarded value. All values forwarded from nonfaulty processors are of the form $R(x)$ for some value x. By another counting argument we see that the nonfaulty processors form a majority of the non-manifestly faulty processors, and therefore the values $R(x)$ forwarded by them win the majority vote, and after applying ***UnR***, all nonfaulty receivers settle on the value actually sent by the transmitter. □

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

Proof: This theorem is also proved by induction on m. In the base case there can be no arbitrary faulty processors, since $m \geq a$ and $m = 0$, so by the previous theorem we have the result. In the inductive step there are two cases: when the transmitter is arbitrary-faulty, and otherwise. In the latter case, again the previous theorem is sufficient. In the former case, by a counting argument the inductive hypothesis can be applied to show that all nonfaulty processors arrive at the same set of values before the hybrid majority vote is taken. If there is a majority all nonfaulty processors will agree on that value, and if there is no majority, all nonfaulty processors will agree on the functionally determined value. Whatever that value, all nonfaulty processors will arrive at consistent values after applying the function *UnR*. □

4 Formal Specification

The two-page formal specification of OMH in PVS is given at the end of this section.[2] The specification language of PVS is a strongly typed higher-order logic. The language supports modularity and reuse by means of parameterized *theories*, and has a rich type system, including the notions of *predicate subtype* and *dependent type*. These makes type checking undecidable (i.e., it can require theorem proving), but provides many benefits (e.g., allowing many functions that are partial in other treatments to become total functions on precisely specified domains). A PVS theory consists of a theory name, a parameter list, and a sequence of *declarations*, which provide names for types, constants, (logical) variables, axioms, and formulas. There is a large body of standard theories built into PVS, collectively referred to as the *prelude*.[3]

Here we will focus on a few interesting aspects of the OMH specification; a detailed English explanation of entire specification is given in [5].[4] The most important parts of a PVS specification are the parameters, assumptions, imported theories, and axioms. Only if all of these match the intended use is the theory potentially useful. Validating a specification against its intended interpretation is as important as verifying the proofs in that specification.

The theory OMH takes several parameters: m, the maximum number of rounds; n, the number of processors; T, the type of possible data values; *error*, a particular

[2] PVS uses ascii input representations; the mathematical symbols appearing in the specification are generated by the LaTeX-printer of PVS.

[3] The PVS system and full documentation for Sun SPARC workstations are available by anonymous FTP from `ftp.csl.sri.com` in directory `/pub/pvs`.

[4] This can be obtained by anonymous FTP from directory `/pub/reports`.

element of T; and the two functions explained earlier, R and *UnR*. There are two assumptions made about the functions R and *UnR*: tagged values are not mistaken for error values, and untagging a tagged value results in the original value. Four theories (not shown here) are brought in by the keyword IMPORTING: ***finite_cardinality***, which defines the set cardinality function $|\,|$; ***filters***, which defines the function ***filter***; ***card_set***, which supplies a set of lemmas regarding the previous two theories, such as "a set is empty if and only if its cardinality is zero"; and ***hybrid_mjrty***, which defines and proves correct a version of the Boyer-Moore MJRTY algorithm [2] that ignores ***error*** values (see [5]). The first three of these four imported theories are from the prelude of standard theories.

The OMH 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. The type ***fcuset*** represents sets of ***fcu***s, and is specified in terms of the predefined type-constructor ***setof***, also from the PVS prelude. The type ***fcuvector*** is specified as the type of functions from ***fcu***s to T. The type ***statuses*** is defined to be an enumeration of exactly four constants, corresponding to the four categories of behavior: ***arbitrary***, ***symmetric***, ***manifest***, and ***nonfaulty***. The function ***status*** returns the status of a given processor (i.e., ***fcu***); this implicitly enforces our notion that a processor not change status during execution of the agreement protocol.

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 (nonfaulty) 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 only AXIOMs of this theory are those involving ***send***, which models messages between processors and is specified here as a function that 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 nonfaulty processor. The result actually received is irrelevant if the receiver is faulty because the values passed on by faulty receivers are not assumed to be related to those received. The first axiom says that a nonfaulty processor sends only correct values. The second axiom says that a manifest-faulty processor always delivers values that are recognized as erroneous by receivers. The third axiom says that a symmetric-faulty processor sends the same value to all nonfaulty receivers, although that value is otherwise unconstrained (i.e., it may be any possible value, including those that are recognized as erroneous). Nothing is specified for the behavior of arbitrary-faulty senders. A deficiency of this specification is that, because *send* is a function, even arbitrarily faulty processors are constrained to be consistent from one round to the next. This fact is not exploited in the proof, and our colleague Shankar has formalized the classic 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 presentation.

Although OMH is conceived as a distributed, concurrent algorithm, its correctness argument need not involve a model of distributed computation: the manner by which values are communicated from one processor to another, and the times and orders in which they are received, can be ignored at the level of abstraction concerned with correctness of the essential algorithm. Consequently, we specify OMH as a simple recursive function: $\mathit{OMH}(G, r, t, \mathit{caucus})(p)$ is the value that processor p ends up with when processor G uses an r-round algorithm to distribute the value t to the processors in the set *caucus*. The definition of *OMH* says that this is just $\mathit{send}(t, G, p)$ in the base case (i.e., $r = 0$), or if $p = G$. Otherwise, p adopts the *UnR* of the hybrid majority of the set of values p receives from all other receivers when they employ *OMH* with one less round.

The two main theorems are first defined as predicates on the number of rounds, then lemmas assert that the predicates hold for all r, and finally theorems give the result in the form desired.

The first theorem, *Validity_final*, instantiates the inductive validity property with the full set of processors. We claim this captures the intent behind the semiformal statement of Theorem 1. We also claim that *Agreement_final* captures the intent behind the semiformal statement of Theorem 2.

```
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, v_1, v_2 : VAR fcuvector
caucus : VAR fcuset
r : VAR rounds
IMPORTING finite_cardinality[fcu, n, identity[fcu]],
          filters[fcu],
          card_set[fcu, n, identity[fcu]],
          hybrid_mjrty[T, n, error]

statuses : TYPE = {arbitrary, symmetric, manifest, nonfaulty}
status : [fcu → statuses]
a(z) : bool = arbitrary(status(z))
s(z) : bool = symmetric(status(z))
c(z) : bool = manifest(status(z))
g(z) : bool = nonfaulty(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
  |caucus| = |as(caucus)| + |ss(caucus)| + |cs(caucus)| + |gs(caucus)|

send : [T, fcu, fcu → T]
send1 : AXIOM g(p) ⊃ send(t, p, q) = t
send2 : AXIOM c(p) ⊃ send(t, p, q) = error
send3 : AXIOM s(p) ⊃ send(t, p, q) = send(t, p, z)
send_lemma : LEMMA ¬ a(p) ⊃ send(t, p, q) = send(t, p, z)

HMajority(caucus, v) : T = proj_1(hybrid_mjrty(caucus, v, n))

HMajority1 : LEMMA
  |gs(caucus)| > |as(caucus)| + |ss(caucus)|
     ∧ (∀ p : g(p) ∧ p ∈ caucus ⊃ v(p) = t)
       ∧ t ≠ error ∧ (∀ p : c(p) ∧ p ∈ caucus ⊃ v(p) = error)
     ⊃ HMajority(caucus, v) = t
```

```
HMajority2 : LEMMA
  $(\forall\, p :\ p \in \text{caucus} \supset v_1(p) = v_2(p))$
     $\supset$ HMajority(caucus, $v_1$) $=$ HMajority(caucus, $v_2$)

OMH($G, r, t$, caucus) : RECURSIVE fcuvector =
    IF $r = 0$
    THEN $(\lambda\, p :$ send($t, G, p$))
    ELSE $(\lambda\, p :$ IF $p = G$
                 THEN send($t, G, p$)
                 ELSE UnR(HMajority(caucus $- \{G\}$,
                                         $(\lambda\, q :$ OMH($q, r-1$, R(send($t, G, q$)), caucus $- \{G\}$)($p$))))
                 ENDIF)
    ENDIF
  MEASURE $(\lambda\, G, r, t$, caucus $\rightarrow$ nat : $r)$

Validity_Prop($r$) : bool =
$\neg\, a(q) \wedge p \in$ caucus $\wedge\ q \in$ caucus
   $\wedge$ |caucus| $> 2 \times$ (|as(caucus)| + |ss(caucus)|) + |cs(caucus)| + $r$
   $\supset$ OMH($q, r, t$, caucus)($p$) = send($t, q, p$)

Validity : LEMMA Validity_Prop($r$)

Validity_Final : THEOREM
  $g(p) \wedge \neg\, a(G) \wedge 2 \times |a| + 2 \times |s| + |c| + m < n$
   $\supset$ OMH($G, m, t$, fullset[fcu])($p$) = send($t, G, p$)

Agreement_Prop($r$) : bool =
  $g(p) \wedge g(q) \wedge p \in$ caucus $\wedge\ q \in$ caucus $\wedge\ z \in$ caucus
   $\wedge$ |caucus| $> 2 \times$ (|as(caucus)| + |ss(caucus)|) + |cs(caucus)| + $r$
   $\wedge\ r \geq$ |as(caucus)|
  $\supset$ OMH($z, r, t$, caucus)($p$) = OMH($z, r, t$, caucus)($q$)

Agreement : LEMMA Agreement_Prop($r$)

Agreement_Final : THEOREM
  $g(p) \wedge g(q) \wedge |a| \leq m \wedge 2 \times |a| + 2 \times |s| + |c| + m < n$
   $\supset$ OMH($G, m, t$, fullset[fcu])($p$) = OMH($G, m, t$, fullset[fcu])($q$)
  END omh
```

5 Formal Verification

The interactive theorem prover of PVS requires all major decisions such as introductions of lemmas and applications of induction to be suggested by the user. PVS then tries to generate the correct instance of a lemma or induction scheme automatically, using pattern matching on the evident formulas. PVS provides arithmetic decision procedures, rewriting, and other automation for the lower levels of deduction, thereby allowing the user to focus on the construction of formal proofs at a relatively high level.

For example, the PVS proof of the lemma *send_lemma* comprises 14 user-supplied steps. This lemma states that all non-arbitrary-faulty processors exhibit symmetric sending behavior. The PVS proof introduces all three *send* axioms, and the definition of *statuses*. PVS chose correct instantiations for the first application of these lemmas (sending to q), but the user was required to suggest the second instance of *send1* and *send2* (sending to z). The application of the ground decision procedures then completed the proof.

The most intellectually challenging proof constructed for this specification is that of *Validity*. After some experimentation with alternative specifications, including constructing partial failed proofs of this lemma for alternative, flawed, versions of the algorithm OMH, the first proof of *Validity* was constructed in a few hours. The polished proof consists of 80 user-suggested steps, 15 of which are invocations of the ground decision procedures. The proof contains 13 invocations of lemmas and axioms, most of them basic lemmas from the prelude and axioms from the OMH theory. Note that the informal proof of Theorem 1 simply says "by a counting argument..." where in the formal proof several lemmas such as "a set is empty if and only if its cardinality is zero" are brought in explicitly, and the ground decision procedures are invoked. Some lemmas require case splitting on the status of a processor (arbitrary, symmetric, manifest, or nonfaulty). The result is a condensed proof description approximately two pages long, and a full prettyprinted proof transcript over 100 pages long.

formula name	user-supplied steps	number of inductions	uses of assert
fincard_all	30	1	7
send_lemma	14	0	1
Validity	80	1	15
Agreement	73	1	13
Validity_final	36	0	4
Agreement_final	51	0	4
HMajority1	78	2	16
HMajority2	20	1	3

The table above summarizes some gross measures of the size and difficulty of constructing proofs for the lemmas and theorems of this specification. The first column is 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. The number of uses of `ground` and `assert` roughly corresponds to the number of significant branches in a proof.

The critical measure, however, for specification and verification tasks is 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 intuitively plausible, and a change in notation for expository purposes. Producing the report [5] and this paper took far more time than the formal specification and verification combined.

When only manifest faults are present, the constraints in Theorems 1 and 2 suggest that the number of faults that can be tolerated is inversely related to the number of rounds. Alternative analysis shows that this is not so and that OMH is optimal in this case. We have formally verified these special cases [5].

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

6 Discussion

We discovered the flaws in Algorithm Z and in our early versions of Algorithm OMH by attempting informal and formal verifications of those algorithms. This experience is consistent with other verification efforts: much of the effort is spent discovering and repairing flaws in a specification, algorithm, or proof that were previously thought to be correct [12, 13]. Although our experience indicates that formal verification is an effective debugging technique, it is undeniably an expensive one.

An obvious alternative is testing: our specifications of these algorithms could be translated into a functional language where they can be run on a variety of test cases. However, testing is inherently partial, and with recursively defined algorithms such as OMH it is difficult to select the important test cases.

Between testing and conventional verification lie *state-exploration* methods. These methods resemble testing in that they are automatic; they resemble verification in that they are formal methods. State-exploration methods systematically enumerate all the states of a finite-state algorithm and test whether certain predicates hold at those states. What makes state exploration effective are recent techniques for handling huge numbers of states in an efficient manner. As it stands, OMH is not amenable to state exploration: it has far too many states. But for debugging, it can be useful to examine highly simplified versions of the problem [3]: for example, the cases $m = 1$, $n \leq 6$, and a very small set of data values—E, $R(E)$, and two distinct "good" values seem sufficient to detect all the errors in all the variants we considered. Theorem proving and state exploration could be combined to prove that some cases are redundant: the case where the first receiver is the only faulty processor is very similar to the case when receiver three is the only faulty processor.

While state exploration might be more economical than conventional formal verification for debugging, correctness of the general algorithm requires full formal verification. Our experience using PVS for this purpose has been very positive. Although still a young prototype system, it has two very important features: a rich specification language, and an effective theorem prover. These attributes usually trade off against each other, since expressive and flexible specification languages tend to be difficult to automate. We have found the paradigm of semi-interactive theorem proving very productive: the user supplies direction for the proof, and the system performs a great deal of the routine manipulation. Putting the user directly in the proving loop requires the current state of the proof to be displayed in a comprehensible manner. (Conversion to clausal form or Skolemization are thus unacceptable, and not used in PVS.) We have found the sequent-calculus presentation used in PVS quite acceptable, but are working on techniques to improve the quality of the representations employed.

7 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 important elements of fault-tolerant architectures is quite modest. In this paper we have presented the formal verification of a new algorithm for Byzantine Agreement under a hybrid fault model. We applied PVS to this domain, discovering errors in published proofs and in a proposed algorithm.

A crucial tool in our detection of the flaws in Algorithm Z and our own early algorithms 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. It is worth repeating that no formal verification proves any system "correct." At most, a model of some aspects of the system is shown to satisfy a specification, or shown to exhibit certain properties. The fidelity of the modeling, and the utility of the specification or properties proved, must be established by informal methods. 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 tirelessly skeptical theorem prover.

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 [8]. These tools (and others that may be of similar effectiveness) are freely available, and 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 [9], 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 extensions to the OMH algorithm and its analysis. We also plan to formally verify a modified version of the Interactive-Convergence Algorithm for clock synchronization using a hybrid fault model (we have already formally verified the standard algorithm [12], and have an informal analysis of a hybrid version). We also plan to continue the development of PVS, improving the ground decision procedures and adding state exploration tools.

Acknowledgments: PVS was constructed by our colleagues Sam Owre and Natarajan Shankar. We have had fruitful discussions with them and with Michelle McElvany-Hugue of Allied-Signal on related topics.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. Technical Report SRI-CSL-93-02, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993.

[6] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993. IEEE Computer Society. To appear.

[7] 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.

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

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

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

[11] 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.

[12] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *SIGSOFT '91: Software for Critical Systems*, pages 1–15, New Orleans, LA, December 1991. Expanded version to appear in *IEEE Transactions on Software Engineering*, 1993.

[13] 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*, pages 217–236, Nijmegen, The Netherlands, January 1992. Volume 571 of *Lecture Notes in Computer Science*, Springer Verlag.

[14] 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.