Formal Verification of an Oral Messages Algorithm for Interactive Consistency[1]

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

Rushby@csl.sri.com
Phone: +1 (415) 859-5456 Fax: +1 (415) 859-2844

Technical Report CSL-92-01
July 1992
Also available as NASA Contractor Report 189704, October 1992

[1]This research was supported by the National Aeronautics and Space Administration under contract NAS1 18969, monitored by NASA Langley Research Center.

Abstract

We describe the formal specification and verification of an algorithm for Interactive Consistency [12] based on the Oral Messages algorithm for Byzantine Agreement [9]. We compare our treatment with that of Bevier and Young [2,3], who presented a formal specification and verification for a very similar algorithm. Unlike Bevier and Young, who observed that "the invariant maintained in the recursive subcases of the algorithm is significantly more complicated than is suggested by the published proof" and who found its formal verification "a fairly difficult exercise in mechanical theorem proving," our treatment is very close to the previously published analysis of the algorithm, and our formal specification and verification are straightforward.

This example illustrates how delicate choices in the formulation of a problem can have significant impact on the readability of its formal specification and on the tractability of its formal verification.

Contents

List of Figures

Chapter 1

Introduction

Fault tolerant systems, such as those used in digital flight control, require a way to ensure that the replicated processors all work on the same input values. For example, each processor may sample different sensors (or the same sensor at different times) and thereby obtain different estimates of some external value; these different estimates need to be combined into a single consensus value that is the same for all processors. By starting with the same inputs, all correctly working processors should then compute the same outputs and faults can be masked using exact-match majority voting.

The problem of deciding on a single consensus value can be broken into two stages. In the first stage, the processors exchange their private data values among themselves. At the end of this stage, each processor has a vector giving the data values of all the other processors; if there are no faults, these vectors will be identical on all processors. The second stage may then comprise any data conditioning, selection, or averaging algorithms whatever: provided all processors run the same algorithms, and start with the same vectors, they will end up with the same consensus values.

We are interested in the first stage of this process, and with ensuring that it performs reliably in the presence of arbitrary faults. (The worst kinds of fault are "asymmetrical" ones where a faulty processor communicates different values to different processors, potentially causing nonfaulty processors to disagree among themselves.) This problem of reaching agreement in the presence of faults was first posed, named, and solved by Pease, Shostak, and Lamport in 1980 [12]. They named the problem that of achieving "Interactive Consistency." In 1982, the same authors developed their analysis further, and reformulated it as the "Byzantine Generals Problem" [9]; they named a revised version of the algorithm from their earlier paper the "Oral Messages" algorithm. The principal difference between the Interactive Consistency and Byzantine Generals problems is that the former is concerned with the reliable exchange of values among all the participants, whereas the latter is concerned with the reliable communication of a value from a distinguished participant (called the "General") to all the others (who are called "lieutenants"). In practical applications, it is the Interactive Consistency formulation that is appropriate, but the colorful metaphor of the Byzantine Generals has proved so memorable that this formulation is better known; indeed, the whole field of algorithm design for agreement in the presence of faults has become known as that of "Byzantine Agreement," and the asymmetrical kind of fault mentioned earlier has become known as a "Byzantine fault."

A problem related to Interactive Consistency is Byzantine fault-tolerant clock synchronization [8]. In 1988, we formally verified the "Interactive Convergence" algorithm for this problem [8, Algorithm CNV] and found that the published analysis of this algorithm was incorrect in a number of details [15,16]. Our colleague Shankar has formally verified the generalized clock synchronization paradigm of Schneider [18] and similarly found a number of small errors [19,20]. In both cases, the formal verification led to improved and simplified presentations of the informal justifications for the correctness of the algorithm concerned. We have often wondered whether formal verification of the Oral Messages algorithm for Byzantine Agreement would yield similar benefits, and have been curious to know how difficult the formal verification of this algorithm would be, compared to the clock synchronization algorithms.

In 1990, a formal verification of the Oral Messages Algorithm was published by Bevier and Young [3] as part of the documentation of a more substantial exercise in which they also verified the design of a circuit to perform the algorithm, and the theorem that the fault-tolerance of the Oral Messages Algorithm is optimal among its class of algorithms.

Bevier and Young described the algorithm as "quite difficult" and have indicated elsewhere that development of its formal verification (using the Boyer-Moore prover [4]) took them about a month. We found this surprising, since the published journal proof for the correctness of the Oral Messages algorithm [9, page 390] is short (less than a page) and straightforward. The time taken may be explained by Bevier and Young's observation [3, page 1] that their machine-checked proof

> "...elucidates several issues which are treated rather lightly in the published version of the proof. In particular, the invariant maintained in the recursive subcases of the algorithm is significantly more complicated than is suggested by the published proof."

After careful study of Bevier and Young's presentation, however, we were unable to persuade ourselves that their claim of suppressed complexity in the published journal proof is justified. On the contrary, we continued to find the journal description and proof more compelling than their formal presentation. In order to resolve our doubts, we decided to undertake a separate formal verification using our EHDM system [17].

There are relatively few examples of interesting or difficult verifications undertaken by more than one group, or using more than one system for formal specification and verification. Bill Young's comparison of Z and Gypsy [24] and the 12-way comparison reported by Jeannette Wing [23] are concerned solely with specification. Rather more interesting are David Basin and Matt Kaufmann's comparison of two verifications of the finite Ramsey theorem [1], and Bill Young's duplication [25] of our verification [15, 16] of a clock synchronization algorithm [8].

One reason for the paucity of comparisons using substantial or difficult examples is that only a handful of verification systems are capable of undertaking such examples, and the developers and users of those systems are fully engaged in their own lines of enquiry. When they can be performed, however, such comparisons are very useful, since they provide the only reasonable way to compare claims for "readability" or "expressiveness" in specification languages, and "power" or "effectiveness" in verification environments.

Comparative studies can be undertaken at several different levels: two different systems can be used to proof-check the same verification (cf. the clock-synchronization example mentioned above); two different verifications can be performed for the same specification; two different formalizations can be developed for the same specification (cf. the study reported by Jeannette Wing); or two completely separate formal developments can be performed for a single problem. Different lessons are likely to be learned from these different levels of comparison: when one tool or notation is simply substituted for another, we may learn something about the ability of the second to duplicate the results of the first on its "home ground," but we will not learn how the problem might have been approached differently had the second tool or notation been used from the start; and when two independent developments are undertaken, we may learn more about the problem-solving approaches of the individuals concerned than about the tools employed.

The experiment described here is of the latter kind, and it may be that the main conclusion to be drawn concerns the considerable impact that apparently small changes in the formulation of a problem can have on the tractability of its formal verification. On the other hand, this example also invites speculation on the beneficial influence that an expressive specification language and a direct approach to proof may have in the development of felicitous formulations of interesting algorithms.

Chapter 2

Informal Overview

In this section we briefly review the Interactive Consistency (IC) and Byzantine Generals (BG) problems, and the "Original" (OA) and Oral Messages (OM) algorithms for solving them. We follow the presentations of Pease, Shostak, and Lamport [9,12] very closely.

2.1 Interactive Consistency

Consider a set of n isolated processors, of which some may be faulty. It is not known which processors are faulty, nor how many, nor what behavior may be exhibited by faulty processors. Suppose also that each processor p has some private value v_p (such as its reading of some sensor). The problem is to devise an algorithm that will allow each processor p to compute a vector V_p of values, in which, for each processor r, $V_p(r)$ is p's estimate of r's private value, satisfying the following conditions:

IC1: If processors p and q are nonfaulty, then they agree on the value ascribed to any other processor r; that is: $V_p(r) = V_q(r)$.

IC2: If processors p and r are nonfaulty, then the value ascribed to r by p is indeed r's private value; that is, $V_p(r) = v_r$.

2.1.1 Oral Messages

There are many variations on the IC and BG problems that differ in the assumptions made about interprocessor communications. For example, whether the processors are fully connected, whether messages can be lost, and whether a faulty processor can forge a message purporting to have come from another. The Oral Messages assumptions are:

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.

An algorithm based on Oral Messages solves the IC problem under these assumptions. The principal difficulty that must be overcome by such an algorithm is that a faulty processor may send different values to different nonfaulty processors, thereby complicating satisfaction of condition IC1. To overcome this, an algorithm will use several "rounds" of message exchange during which processor p tells processor q what value it received from processor r and so on. Of course, if processor p is faulty, it may "lie" about the value it received from processor r. By making sufficiently many rounds, an algorithm can defeat this threat.

2.1.2 The Original Algorithm

The original algorithm [12, page 230], which we will abbreviate as OA, is parameterized by n, the number of processors, and m (where $n \geq 3m+1$), the maximum number of faulty processors. The following description of OA is taken verbatim from [12, page 230] (except that we have changed V to v).

> "Let P be the set of processors and v a set of values. For $k \geq 1$, we define a *k-level scenario* as a mapping from the set of nonempty strings (possibly having repetitions) over P of length $\leq k+1$, to v. For a given k-level scenario, σ and string $w = p_1 p_2 \ldots p_r$, $2 \leq r \leq k+1$, $\sigma(w)$ is interpreted as the value p_2 tells p_1 that p_3 told p_2 that p_4 told p_3... that p_r told p_{r-1} is p_r's private value. For a single-element string p, $\sigma(p)$ simply designates p's private value v_p. A k-level scenario thus summarizes the outcome of a k-round exchange of information. (Note that if a faulty processor lies about who gave it information, this is equivalent to lying about a value it was given.) Note also that for a given subset of nonfaulty processors, only certain mappings are possible scenarios; in particular, since nonfaulty processors are always truthful in relaying information, a scenario must satisfy
>
> $$\sigma(pqw) = \sigma(qw)$$
>
> for each nonfaulty processor q, arbitrary processor p, and string w.
>
> "The messages a processor p receives in a scenario σ are given by the restriction σ_p of σ to strings beginning with p. The procedure we present now for arbitrary $m \geq 0$, $n \geq 3m+1$, is described in terms of p's computation for a given σ_p, of the element of the interactive-consistency vector corresponding to each processor q (i.e., $V_p(q)$). The computation is as follows:

1. If for some subset Q of P of size $> (n+m)/2$ and some value ν, $\sigma_p(pwq) = v$ for each string w over Q of length $\leq m$, p records ν.
2. Otherwise the algorithm for $m-1$, $n-1$ is recursively applied with P replaced by $P - \{q\}$, and σ_p by the mapping $\hat{\sigma}_p$ defined by

$$\hat{\sigma}_p(pw) = \sigma_p(pwq)$$

for each string w of length $\leq m$ over $P - \{q\}$. If at least $\lfloor (n+m)/2 \rfloor$ of the $n-1$ elements in the vector obtained in the recursive call agree, p records the common value; otherwise p records NIL.

Note that $\hat{\sigma}_p$ corresponds to the m-level subscenario of σ in which q is excluded and in which each processor's private value is the value it obtains directly from q in σ."

We expect that many readers will share our opinion that this description of OA is a challenge to comprehension. The argument for its correctness [12, page 231] is similarly hard to follow. The original authors also may have considered the presentation somewhat difficult, for a couple of years after the original publication they reformulated the problem, the algorithm, and the argument for its correctness. The revised presentation was couched in the metaphor of "Byzantine Generals" and is described in the next section.

2.2 Byzantine Generals

As mentioned earlier, BG differs from IC in that there is a distinguished processor called the *General* whose value is to be communicated to all other processors (called *lieutenants*).[1] Again, there are n processors in total, of which some (possibly including the General) may be faulty. The General has some "order" v and the problem is to devise an algorithm that will allow each Lieutenant p to compute an estimate ν_p of the General's order satisfying the following conditions:

BG1: If Lieutenants p and q are nonfaulty, then they agree on the value ascribed to the General; that is $\nu_p = \nu_q$.

BG2: If the General is nonfaulty, then every nonfaulty lieutenant has the correct order; that is $\nu_p = v$.

We have renamed these conditions BG1 and BG2 to distinguish them from the corresponding conditions of the IC case.

[1] Lamport, Shostak and Pease [9] often speak of the "Commanding General," and refer to the others as the "lieutenant generals."

The Oral Messages (OM) algorithm solves the BG problem under the same assumptions as OA; it can be regarded as a substantial reformulation of OA, rather than an independent algorithm. In order to distinguish the BG version of the algorithm from the IC version to be introduced later, we denote them OMBG and OMIC, respectively. The algorithm is characterized by the number of rounds to be made: OMBG(m) is the instance of the algorithm that makes $m+1$ rounds. The following description is taken verbatim from [9, page 388]. Note that under the Byzantine Generals metaphor, faulty processors are called "traitors," and nonfaulty ones are "loyal." First we describe the simplest case, OMBG(0):

OMBG(0)

1. The General sends his value to every lieutenant.
2. Each lieutenant uses the value he receives from the General, or uses the value *retreat* if he receives no value.

Now we can describe the general case.

OMBG(m), m $>$ 0

1. The General sends his value to every lieutenant.
2. For each i, let v_i be the value Lieutenant i receives from the General, or else be *retreat* if he receives no value. Lieutenant i acts as the General in Algorithm OMBG($m-1$) to communicate the value v_i to each of the $n-2$ other lieutenants.
3. For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OMBG($m-1$)), or else *retreat* if he received no such value. Lieutenant i uses the value $majority(v_1, \ldots, v_{n-1})$.

2.2.1 The Correctness Argument

The argument for the correctness of OMBG is taken verbatim from [9, page 390]

Lemma 1 *For any m and k, Algorithm OMBG(m) satisfies BG2 if there are more than $2k+m$ participants and at most k traitors.*

Proof: The proof is by induction on m. BG2 only specifies what must happen if the General is loyal. Using A1, it is easy to see that the trivial algorithm OMBG(0) works if the General is loyal, so the lemma is true for $m = 0$. We now assume it is true for $m-1$, $m > 0$, and prove it for m.

In step (1), the loyal General sends a value v to all $n-1$ lieutenants. In step (2), each loyal lieutenant applies OMBG($m-1$) with $n-1$ generals. Since by hypothesis $n > 2k+m$, we have $n-1 > 2k+(m-1)$, so we can apply the induction hypothesis to conclude that every loyal lieutenant gets $v_j = v$ for each loyal Lieutenant j. Since there are at most k traitors, and $n-1 > 2k + (m-1) \geq 2k$, a majority of the $n-1$ lieutenants are loyal. Hence, each loyal lieutenant has $v_i = v$ for a majority of the $n-1$ values i, so he obtains $majority(v_1, \ldots, v_{n-1}) = v$ in step (3), proving BG2. $\square$

Theorem 1 *For any m, Algorithm OMBG(m) satisfies conditions BG1 and BG2 if there are more than $3m$ participants and at most m traitors.*

Proof: The proof is by induction on m. If there are no traitors, then it is easy to see that OMBG(0) satisfies BG1 and BG2. We therefore assume that the theorem is true for OMBG($m-1$) and prove it for OMBG(m), $m > 0$.

We first consider the case in which the General is loyal. By taking k equal to m in Lemma 1, we see that OMBG(m) satisfies BG2. BG1 follows from BG2 if the General is loyal, so we only need verify BG1 in the case the General is a traitor.

There are at most m traitors, and the General is one of them, so at most $m-1$ of the lieutenants are traitors. Since there are more than $3m$ generals, there are more than $3m-1$ lieutenants, and $3m-1 > 3(m-1)$. We may therefore apply the induction hypothesis to conclude that OMBG($m-1$) satisfies conditions BG1 and BG2. Hence, for each j, any two loyal lieutenants get the same value for v_j in step (3). (This follows from BG2 if one of the two lieutenants is Lieutenant j, and from BG1 otherwise). Hence, any two loyal lieutenants get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value $majority(v_1, \ldots, v_{n-1})$ in step (3), proving BG1. $\square$

Chapter 3

Bevier and Young's Verification

Bevier and Young [3] performed a formal specification and verification of the OMBG Algorithm using the Boyer-Moore theorem prover [4]. Insofar as the restrictions of the Boyer-Moore logic allow[1], Bevier and Young's specification and verification follows the published version of Lamport, Shostak and Pease [9] very closely. Since the problem of practical interest is IC rather than BG, they augment their description [3, Section 3.4] with the specification and verification of an additional step that applies OMBG iteratively (with each process in turn taking the role of the General), thereby extending it to a solution for IC.

Bevier and Young specify OMBG in terms of two mutually recursive functions, `vom*` and `voml*`; the former is the main OMBG function, while the latter specifies the iterative application over all lieutenants required in step (2) of the former. In addition, the function `vom0` specifies the base case OMBG(0). These functions are reproduced in Figure 3.1 (taken from [3, Figure 5, page 7]).[2]

Bevier and Young explain these functions as follows [3, pages 6,7]. Note that the function `(send v i j)` denotes the value received when process i sends value v to j.

> "`vom*` is the top-level function which takes as arguments the number `m` of rounds, the General's name `g` and value `v`, a list `l` of lieutenant names, and the vector `vec` in which the message traffic is recorded. It returns a vector in which each lieutenant's position is filled by that lieutenant's view of the General's value. Arriving at this view requires $m-1$ rounds of communication (the call to the `voml*` function) combined (`pair`'d) with the initial round in which the General distributes

[1] The Boyer-Moore logic is an untyped, unquantified first-order logic resembling pure lisp.

[2] In order to satisfy the definitional principle of the Boyer-Moore system, the mutually recursive pair `vom*` and `voml*` are encoded in the actual specification as a single function with a "flag" argument to distinguish the two cases. This version is reproduced in Appendix A, Figure A.1.

Definition

```
(vom0 g v l vec)
   =
(if (listp l)
    (put (car l)
         (send v g (car l))
         (vom0 g v (cdr l) vec))
     vec)
```

Definition

```
(vom* m g v l vec)
   =
(if (zerop m)
    (vom0 g v l vec)
    (votelist
      (pair (vom0 g v l vec)
            (voml* (sub1 m) l (vom0 g v l vec) l vec)
             l)))
```

Definition

```
(voml* m g-list vom0 1 vec)
  =
(if (listp g-list)
    (pair (vom* m (car g-list) (get (car g-list) vom0)
                (delete (car g-list) l) vec)
          (voml* m (cdr g-list) vom0 l vec)
          (delete (car g-list) l))
    (init nil (length vec)))
```

Figure 3.1: Bevier and Young's Specification of the Oral Messages Algorithm

> his value directly (the call to `vom0`), and voting on each element in the resulting map (the call to `votelist`).
>
> "The function `voml*` takes as arguments the number `m` of exchanges, a list `g-list` of names of processes which will serve in turn as the general in this round, a vector `vom0` in which each process's slot is filled with its value sent to it by the General, a list `l` of the other lieutenants, and a vector `vec` in which the message traffic is recorded. It returns a vector in which each lieutenant's name is bound to the *list* of messages that lieutenant has received in this round of message exchanges."

Bevier and Young state that the verification that their specifications of OMBG satisfy BG1 and BG2 is "a fairly difficult exercise in mechanical theorem proving" [3, page 1] but that they "gained considerable insight into the algorithm" from their formalization [3, page 13]. They illustrate the latter point by referring to the published proof of OMBG (reproduced in Section 2.2.1 above) and observing:

> "Though seemingly straightforward, there is a considerable degree of suppressed detail in this proof. In particular, the induction hypothesis refers to what happens *after* each round of message exchange without worrying about the intermediate states which occur *during* each round. In terms of our mutually recursive version of the algorithm, the proof above describes the induction by referring to what happens after each call to `vom*` and simply assumes what happens in the calls to `voml*`.
>
> "What happens in these calls, and what is crucial from the point of view of a fully formal proof, is that there is a rather involved invariant maintained by the algorithm. A key part of this invariant can be stated roughly as follows: after each round of message exchange all of the non-faulty processors agree on a value for the General, that value being the General's actual value. This notion we call *non-faulty agreement*.
>
> "Formulating and proving an appropriate version of the invariant for BG2 was the primary effort in the proof."

The invariant referred to above is reproduced in Figure 3.2. Bevier and Young "do not bother to describe some of the subsidiary concepts such as `non-faulty-value` which are involved in the statement of the invariant" [3, page 14] and do not exhibit the corresponding invariant for BG1, but note that it "is substantially more involved."

Theorem. VOM-IC2-INVARIANT

```
(implies
  (and (setp l)
       (bounded-number-listp l (length vec))
       (member i l)
       (not (faulty i)))
  (if flg
      (implies
        (and (not (member g l))
             (not (faulty g))
             (leq (plus (times 2 (fault-count l)) m)
                  (length l)))
        (equal (get i (vom flg m g v l vec))
               v))
    (implies
      (and (subbagp g l)
           (equal (length v) (length vec))
           (lessp (plus (times 2 (fault-count l)) m)
                  (length l))
           (non-faulty-agreement (non-faulty-value g v)
                                 g v))
      (not (lessp (occurrences
                    (non-faulty-value g v)
                    (get i (vom flg m g v l vec)))
                  (if (member i g)
                      (sub1 (good-count g))
                    (good-count g)))))))
```

Figure 3.2: Bevier and Young's "Invariant" for BG2

Chapter 4

Specification and Verification in EHDM

One source of complexity in both the specification and verification of Bevier and Young's formulation of OMBG is the need for a pair of mutually recursive functions. An additional burden is the need to perform a second specification and verification in order to connect BG to IC. Both of these difficulties can be avoided by developing a version of OM that solves IC directly. One way to see that this approach is likely to be beneficial is to observe that the iterated recursion inside OMBG is solving an instance of IC: after the General has transmitted his value to all the lieutenants, each of those lieutenants has a private value (the value he received from the General), and the subgoal is for the $n-1$ lieutenants to perform IC on those private values. Each lieutenant will then have an IC vector that gives the value sent by the General to each lieutenant; all nonfaulty lieutenants will have the same IC vector, and selecting the majority value from those vectors will cause each of them to assign the same value to the General.

It follows that a generalization of OMBG from BG to IC should be simpler than OMBG, since the recursive subproblems will be the same as the parent. We will call this generalization the OMIC algorithm. We present the algorithm and the argument for its correctness in the next few pages. All the specifications that follow in this section are taken directly from our formal verification, and are in the language of EHDM [17]; the proof sketches are also taken from our formal verification. The full specification and verification is presented in Appendix C.

We will specify OMIC as a function of three arguments: m the number of rounds, v a vector giving the private values of each processor, and *caucus* the set of processors participating in the algorithm. Processors are represented by natural numbers in the range $0 \ldots n-1$, and vectors are functions from processors to values (of some uninterpreted type T). OMIC will return a "vector" of vectors: that is a function from processors to vectors. Thus $\mathrm{OMIC}(m, v, \mathit{caucus})(p)$ will be the IC vector of processor p following the OMIC algorithm, and OMIC$(m, v, \mathit{caucus})(p)(q)$ will be p's opinion of q's private value (i.e., of $v(q)$). Notice that we are using higher-order functions (i.e., functions whose values are functions) here. We have found higher-order constructions very convenient in several specifications that we have undertaken (see for example [13]).[1]

In preparation for formally specifying OMIC, we first state its behavior for the case $m = 0$. In this and the formulas that follow, free variables are treated as universally quantified at the outermost level, and we do not generally identify the types of the variables appearing in these formulas (see the full specification in Appendix C for these subsidiary declarations).

$$\begin{aligned}&\mathrm{OMIC}(0, v, \mathrm{caucus})(p)(q)\\ &\quad = \textbf{if } p \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \textbf{ then } \mathrm{send}(v(q), q, p) \textbf{ else } \mathrm{undef} \textbf{ end if}\end{aligned}$$

Here, *undef* is some arbitrary value and $\mathit{send}(v(q), q, p)$ is, as in Bevier and Young's formulation a function that represents the value received by p when q sends it the value $v(q)$. Our requirement on OMIC in the case $m = 0$ simply states that if p and q are both participants to the algorithm (i.e., both in the set *caucus*), then p's opinion of q's private value $v(q)$ following the algorithm should be $\mathit{send}(v(q), q, p)$.

The property assumed of *send* is captured in the following axiom

$$\text{send\_ax: } \textbf{Axiom } \mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{send}(t, q, p) = t$$

where $\mathit{ok}(p)$ is the predicate that asserts that processor p is nonfaulty. (We regard a processor that is faulty at any point in the algorithm as being faulty throughout.) Essentially, this axiom captures Assumption A1 of oral messages. Notice that if either p or q are faulty, we know nothing whatever about the value $\mathit{send}(t, q, p)$. Well, not exactly nothing: we do know that its value is functionally determined by t, p, and q. Thus, if q were to send t to p in a later round, the value received would be the same as in this round, whatever the fault-status of the processors concerned. This may not be realistic if p or q are faulty, so we will reformulate *send* to take the round number as an argument: $\mathit{send}(r, t, q, p)$ then represents the value received by p when q sends it the value t in round r. The round number does not affect the transmission when nonfaulty processors are involved:

$$\text{send\_ax: } \textbf{Axiom } \mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{send}(r, t, q, p) = t$$

[1] Higher-order functions are also used in EHDM to specify set operations, which appear frequently in this specification. Sets are specified as their characteristic predicates in EHDM and the operation that, for example, removes a processor from a set of processors has the specification

$$\mathrm{caucus} - \{q\}\text{: } \mathrm{function}[\mathrm{set}, \mathrm{processors} \rightarrow \mathrm{set}] = (\lambda\, \mathrm{caucus}, q : \mathrm{caucus}\ \textbf{with}\ [(q) := \mathrm{false}])$$

The only effect and purpose of this modified treatment of the *send* function is to make it more clear that no assumptions at all are made about values communicated when either the sender or receiver is faulty.[2]

The specification of the behavior of OMIC in the case $m = 0$ needs to be adjusted accommodate the changed functionality of *send*:

$$\begin{array}{l} \mathrm{OMIC}(0, v, \mathrm{caucus})(p)(q) \\ \quad = \textbf{if } p \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \textbf{ then } \mathrm{send}(0, v(q), q, p) \textbf{ else } \mathrm{undef} \textbf{ end if} \end{array}$$

For the case $m = r$, $r > 0$, we require that p's opinion of q's private value should be $send(r, v(q), q, q)$ if $p = q$,[3] otherwise it should be the majority value in p's IC vector, 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. Thus we require

$$\begin{array}{l} r > 0 \supset \mathrm{OMIC}(r, v, \mathrm{caucus})(p)(q) \\ \quad = \textbf{if } p \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \\ \qquad \textbf{then } \textbf{if } p = q \\ \qquad\quad \textbf{then } \mathrm{send}(r, v(q), q, q) \\ \qquad\quad \textbf{else } \mathrm{maj}(\mathrm{caucus} - \{q\}, \mathrm{OMIC}(r-1, \mathrm{distr}(r, v(q), q), \mathrm{caucus} - \{q\})(p)) \\ \qquad \textbf{end if} \\ \qquad \textbf{else } \mathrm{undef} \\ \qquad \textbf{end if} \end{array}$$

Here, $distr(r, v(q), q)$ is simply a function that uses *send* in round r to distribute the value $v(q)$ from q to every other process:[4]

$$\mathrm{distr}\colon \mathrm{function}[\mathit{rounds}, T, \mathrm{processors} \rightarrow \mathrm{vector}] = (\,\lambda\, r, t, p : (\,\lambda\, z : \mathrm{send}(r, t, p, z)))$$

---

[2] Even the modified formulation of *send* is not free of the suggestion that the value delivered by a faulty processor is functionally determined. The only way to completely remove this taint is to use a relational specification for *send*, with the interpretation that $\mathrm{send}(t_1, q, p, t_2)$ is true if p could possibly receive t_2 when q sends it t_1, and the axiom:

$$\mathrm{send\_ax}\colon \textbf{Axiom } \mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{send}(t, q, p, t)$$

The problem with this reformulation is that it has ramifications throughout the specification, requiring the definition of the Oral Messages algorithm itself, as well as several subsidiary functions, to become relations also. Our colleague Shankar has recently developed such a specification and verification of OMIC, but here we prefer to stay with the simpler, if slightly flawed, functional definition of *send*.

[3] 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.

[4] It might be less "wasteful" to add the set of recipient processors (i.e., $caucus - \{q\}$) as an additional argument to *distr*, rather than have the value sent to every process. This sort of "economy" would be important in an implementation of the algorithm, but would clutter the specification and proof.

The function *maj* takes a set *caucus* of processors, and a vector v, and computes the majority value (if any) in that vector over that set. Actually, requiring this function to be implemented by a majority vote overspecifies the problem. All that is really required is specified in the following axiom, which states 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 *maj* function. Notice that 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 [9, page 388]).

majax: **Axiom**
$|\text{caucus}| > 2 * |\text{faulty\_members}(\text{caucus})| \wedge (\forall\, p : \text{ok}(p) \wedge p \in \text{caucus} \supset v(p) = t)$
$\supset \text{maj}(\text{caucus}, v) = t$

The function application *faulty_members*(*caucus*) that appears here is the set of faulty (i.e., not *ok*) processors in the set *caucus*:

faulty_members: function[set → set] = $(\lambda\, m_1 : (\lambda\, z : z \in m_1 \wedge \neg\text{ok}(z)))$

Vertical bars denote the cardinality function. The only properties we require of this function are captured in the following axioms.

$|\star 1|$: function[set → nat]

non_empty_ax: **Axiom** $(\exists\, p : p \in m_1) \Leftrightarrow |m_1| \neq 0$

card_remove_ax: **Axiom** $z \in m_1 \supset |m_1 - \{z\}| = |m_1| - 1$

A second requirement on the *maj* function is that its value depends on only those elements of the vector corresponding to members of the set *caucus*.

maj_ext: **Axiom**
$(\forall\, p : p \in \text{caucus} \supset v_1(p) = v_2(p)) \supset \text{maj}(\text{caucus}, v_1) = \text{maj}(\text{caucus}, v_2)$

We now return to the specification of OMIC. The two behaviors that were stated above (for the cases $m = 0$, and $m > 0$, respectively) could be specified as axioms defining the function; 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 EHDM 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 measure function *terminates* that simply returns its first argument (i.e., the number of rounds). The final specification for OMIC is given in Figure 4.1.

terminates: function[rounds, vector, set $\rightarrow$ nat] = (λ r, v, caucus $\rightarrow$ nat : r)

OMIC: **Recursive** function[rounds, vector, set $\rightarrow$ function[processors $\rightarrow$ vector]] =

```
( λ r, v, caucus :
    if r = 0
     then ( λ p :
       ( λ q :
            if p ∈ caucus ∧ q ∈ caucus
             then send(r, v(q), q, p)
             else undef
            end if))
     else ( λ p :
       ( λ q :
            if p ∈ caucus ∧ q ∈ caucus
             then  if p = q
               then send(r, v(q), q, q)
               else maj(caucus − {q},
                        OMIC(r − 1, distr(r, v(q), q), caucus − {q})(p))
             end if
             else undef
            end if))
   end if)
 by terminates
```

Figure 4.1: Our specification of the Oral Messages Algorithm

We invite the reader to compare this specification with that of Bevier and Young that was shown in Figure 3.1. Since our specification is pretty-printed (a function performed automatically by EHDM), while Bevier and Young's is given in raw text form, the versions shown in Appendix A, which reproduce the exact text submitted to their respective theorem proving environments, allow more exact comparison.

The Interactive Consistency conditions IC1 and IC2 are easily stated as theorems to be proven:

C1_final: **Theorem**
$\mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{OMIC}(m, v, \mathrm{fullset})(p)(y) = \mathrm{OMIC}(m, v, \mathrm{fullset})(q)(y)$

C2_final: **Theorem** $\mathrm{ok}(p) \wedge \mathrm{ok}(q) \supset \mathrm{OMIC}(m, v, \mathrm{fullset})(p)(q) = v(q)$

where *fullset* is the set of all processors.

As in the informal proof of Section 2.2.1, we begin by proving a lemma similar to IC2. The proof is by induction and in formal verifications it is usually convenient to reformulate the theorem to be proved as a predicate on the induction variable. Here, we call the predicate *C2prop*.

$$
\begin{aligned}
&\text{C2prop: function[rounds} \rightarrow \text{bool]} = \\
&\quad (\lambda\, r : (\forall\, p, q, \mathrm{caucus}, v : \\
&\qquad\quad \mathrm{ok}(p) \wedge \mathrm{ok}(q) \\
&\qquad\qquad \wedge\, p \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \\
&\qquad\qquad\quad \wedge\, |\mathrm{caucus}| > 2 * |\mathrm{faulty\_members}(\mathrm{caucus})| + r \\
&\qquad\quad\; \supset \mathrm{OMIC}(r, v, \mathrm{caucus})(p)(q) = v(q)))
\end{aligned}
$$

The base case of the induction (i.e., *C2prop*(0)) follows by straightforward application of definitions; the inductive step (i.e., $r < m \wedge \mathit{C2prop}(r) \supset \mathit{C2prop}(r+1)$) follows from two lemmas. The first, which asserts that a good processor has the correct opinion of its own value, is straightforward:

ok_self: **Lemma** $\mathrm{ok}(y) \wedge y \in \mathrm{caucus} \supset \mathrm{OMIC}(r, v_2, \mathrm{caucus})(y)(y) = v_2(y)$

The second, which asserts that under certain conditions a good processor forms the correct opinion of the private value of another good processor, is more complex.

ok_others: **Lemma**

$$
\begin{aligned}
&r < m \wedge |\mathrm{caucus} - \{q\}| > 2 * |\mathrm{faulty\_members}(\mathrm{caucus} - \{q\})| \\
&\qquad \wedge\, \mathrm{ok}(y) \wedge \mathrm{ok}(q) \\
&\qquad\quad \wedge\, y \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \\
&\qquad\qquad \wedge\, y \neq q \\
&\qquad\qquad\quad \wedge\, (\forall\, z, v_1 : z \in \mathrm{caucus} \wedge \mathrm{ok}(z) \wedge z \neq q \\
&\qquad\qquad\qquad\qquad \supset \mathrm{OMIC}(r, v_1, \mathrm{caucus} - \{q\})(y)(z) = v_1(z)) \\
&\quad \supset \mathrm{OMIC}(r+1, v_2, \mathrm{caucus})(y)(q) = v_2(q)
\end{aligned}
$$

Verification of this property depends on the *majax* axiom of the *maj* function.

The two lemmas above are sufficient to establish the inductive step for verification of *C2prop*(r); observe that the hypothesis to the inductive step discharges the quantified subexpression in ok_others. The theorem *C2_final* follows straightforwardly from *C2prop*(r) by substitution of m for r and *fullset* for *caucus*, and using the axiom

$$\text{fullset\_card\_ax: } \textbf{Axiom } |\text{fullset}| = n \wedge |\text{faulty\_members}(\text{fullset})| \leq m$$

and the constraint that less than a third of the processors may be faulty:

$$\text{mn\_prop: } \textbf{Formula } 3 * m < n$$

This property is stated as a formula in the *assuming* section of the EHDM module that specifies the theory developed here. It specifies an assumption on the parameters m and n to the module: inside the module, this assumption is treated as an axiom; it must be discharged whenever the module is instantiated.

IC1 is similarly proved by induction, using the following predicate.

$$\begin{array}{l}
\text{C1prop: function[rounds} \rightarrow \text{bool]} = \\
\quad (\lambda\, r : (\forall\, p, q, y, \text{caucus}, v : \\
\qquad\qquad \text{ok}(p) \wedge \text{ok}(q) \\
\qquad\qquad\quad \wedge\, p \in \text{caucus} \wedge q \in \text{caucus} \wedge y \in \text{caucus} \\
\qquad\qquad\qquad \wedge\, |\text{caucus}| > 3 * r \wedge r \geq |\text{faulty_members}(\text{caucus})| \\
\qquad\qquad \supset \text{OMIC}(r, v, \text{caucus})(p)(y) = \text{OMIC}(r, v, \text{caucus})(q)(y)))
\end{array}$$

Again the base case is straightforward; the inductive step has two cases, depending on whether the processor y is faulty or not. The case that it is faulty is dealt with in the following lemma, whose proof is a consequence of the *maj_ext* axiom of the *maj* function.

$$\begin{array}{l}
\text{agree_nok: } \textbf{Lemma} \\
\quad r < m \wedge |\text{caucus}| > 3 * (r+1) \wedge r + 1 \geq |\text{faulty_members}(\text{caucus})| \\
\qquad \wedge\, \text{ok}(p) \wedge \text{ok}(q) \\
\qquad\quad \wedge\, p \in \text{caucus} \wedge q \in \text{caucus} \wedge y \in \text{caucus} \\
\qquad\qquad \wedge\, \neg\text{ok}(y) \\
\qquad\qquad\quad \wedge\, (\forall\, z, v_1 : z \in \text{caucus} - \{y\} \\
\qquad\qquad\qquad \supset \text{OMIC}(r, v_1, \text{caucus} - \{y\})(p)(z) = \text{OMIC}(r, v_1, \text{caucus} - \{y\})(q)(z)) \\
\qquad \supset \text{OMIC}(r+1, v_2, \text{caucus})(p)(y) = \text{OMIC}(r+1, v_2, \text{caucus})(q)(y)
\end{array}$$

The case when y is nonfaulty is treated in the following lemma

agree_ok: **Lemma**

$$r < m \wedge |\text{caucus}| > 3 * (r+1) \wedge r+1 \geq |\text{faulty\_members}(\text{caucus})|$$
$$\wedge \text{ ok}(p) \wedge \text{ok}(q)$$
$$\wedge\, p \in \text{caucus} \wedge q \in \text{caucus} \wedge y \in \text{caucus}$$
$$\wedge \text{ ok}(y)$$
$$\supset \text{OMIC}(r+1, v_2, \text{caucus})(p)(y) = \text{OMIC}(r+1, v_2, \text{caucus})(q)(y)$$

whose proof is a consequence of C2_final.

These two lemmas are sufficient to establish the inductive step for *C1_final*; note that the hypothesis to this step discharges the quantified subexpression in *agree_nok*. *C1_final* follows from *C1prop*(r) in the same way that *C2_final* follows from *C2prop*(r).

The full specification and verification requires development of some "background knowledge." For example, the inductions require a specialized induction scheme that goes from 0 only as far as m. This is stated as the Lemma *round_induct* that is ultimately derived from an axiom for Noetherian induction contained in a standard EHDM library module. The variable *round_prop* is some arbitrary property of rounds that is to be shown to hold for all rounds.

round_prop: **Var** function[rounds → bool]

round_induct: **Lemma**

$$(\text{round\_prop}(0) \wedge (\forall\, r : r < m \wedge \text{round\_prop}(r) \supset \text{round\_prop}(r+1)))$$
$$\supset \text{round\_prop}(s)$$

A full listing of the formal specification is provided in Appendix C, together with EHDM's "proof chain" analysis for C1_final. The latter identifies the axiomatic foundation for our development: this comprises the 6 axioms and the assumption shown here, plus an axiom for induction and another for function extensionality that come from library modules. The subsidiary lemmas required to carry out the formal verification number 23 (plus the two theorems), with another 4 in library modules, and a further 20 typecheck correctness conditions (TCCs—these are proof obligations that must be discharged to ensure type-correctness) that are generated by the typechecker. Only 2 of the TCCs require user-generated proofs; the other 18 are proved automatically.

Chapter 5

Discussion and Conclusion

5.1 Discussion

We have presented the formal specification and verification of an algorithm for Interactive Consistency derived from the Oral Messages algorithm for the Byzantine Generals Problem. Both the specification of the algorithm and the arguments for its correctness are straightforward and closely modeled on those given by Lamport, Shostak and Pease in their journal presentation [9]. Development of the formal specification and its verification in EHDM took about four days. By comparison, Bevier and Young [3], using the Boyer-Moore theorem prover, found formal verification of their version of the algorithm "a fairly difficult exercise in mechanical theorem proving" that occupied them for about a month. We do not know all the complexities that confronted Bevier and Young, and so we cannot identify, much less apportion credit to, all the reasons why we apparently found the verification easier than them.

However, one explanation for these different assessments of the difficulty of the exercise may lie in the different formulations employed for the algorithm. Bevier and Young used the Byzantine Generals formulation, which must be applied iteratively in order to solve the Interactive Consistency problem that is the topic of real interest, and whose recursive subcase likewise requires iteration. This potentially complicates the inductions at the heart of the proof (since the recursive subcase is not simply a smaller instance of the original problem), and the larger verification along with it. The specification of the algorithm may become similarly complicated in this formulation. In contrast, our reformulation of the Oral Messages algorithm solves the Interactive Consistency problem directly, and its recursive subcase is a smaller instance of itself. The formal specification, main inductions, and overall verification are then entirely straightforward.

The lesson here is a variation on the well-known observation that it is sometimes easier to prove a stronger than a weaker theorem when using induction. In particular, it is much easier to prove properties of an algorithm whose recursive subcases are exact replicas of itself: and it may be worth modifying the algorithm, or its requirement, or both, in order to make this so. A related observation, one that we first heard explicitly articulated by our colleague Shankar, is that recursions should always be formulated so that the base case is completely different from the recursive case—since otherwise one may end up verifying substantially the same argument twice.

But our simpler verification cannot be entirely attributed to our reformulation of the Oral Messages algorithm into the IC form, for we have also verified the BG version of the algorithm as considered by Bevier and Young. Our specification of the BG form is not exactly the same as theirs, since our richer specification language allows us to specify the algorithm without the need to simulate a pair of mutually recursive functions (see Appendix B). Nonetheless, our BG formulation is substantially the same as Bevier and Young's and yet its verification is only a little more complex than that of OMIC.[1] However, we must admit that the formulation and verification of the BG version would have been significantly more difficult had we not already performed the IC version; that is specification and verification of IC followed by BG is probably much simpler than tackling BG alone.

But allowing for the advantage we gained by choosing the more tractable approach, we still seem to have found this exercise more straightforward that Bevier and Young, and we attribute some of this to the design decisions embodied in EHDM. The specification language of EHDM is intended to provide a fairly direct and natural means for expressing a variety of mathematical concepts, while retaining a straightforward logical foundation. We were gratified to find that the language helped us to achieve clear descriptions of these tricky algorithms. We find the strong type system and higher-order capabilities particularly helpful in this regard. Identifying the types of the variables and functions involved is a valuable first step in the formulating the specification, since it suggests the ways in which functions should be combined and thereby, in this case, helps determine the shape of the recursion. Higher-order logic allows many ideas to be expressed is a direct manner: thus, we do not require the mutual recursion that complicates Bevier and Young's specification, and we can represent values as functions, without the need to introduce lists.

We have had similar experiences with other specifications that we have undertaken. For example, our formal development in EHDM of a model for fault-masking and transient-recovery in digital flight-control systems [13, 14] was undertaken in parallel with a similarly detailed development using conventional pencil-and-paper mathematical notation [5,6]. The EHDM version took no longer to develop than the other, is more general, is equally readable, and has been fully verified.

The simplifying reformulation of the Oral Messages Algorithm into its IC form is very much the kind of benefit that we strive to obtain from formal methods (see,

[1] The verification, which is available from the author on request, was obtained by modifying the OMIC version, and took about a man-day to produce.

for example, our improved argument for the correctness of the Interactive Convergence clock synchronization algorithm [15, 16]). We are strongly of the opinion that formal methods must contribute to, and cannot stand apart from, established and informal practices in software and hardware engineering. Thus, specifications must be readable by others than their authors, and formal verifications must yield a chain of argument that can be presented to, and will convince, a suitably knowledgeable human reviewer.

5.2 Conclusion

As with other formal developments that we have performed, we derived a significant benefit from this exercise quite apart from the mechanically-checked verification of an interesting argument. Here, the benefit was a reformulation of the Oral Messages Algorithm to solve the Interactive Consistency, rather than the Byzantine Generals problem. This is not only a more useful form of the algorithm in practice, it is rather simpler to specify and to verify. As always, this benefit could have been obtained without formalization, but it was the discipline of formalization that led us to focus on the problem in the manner required.

The simplification produced by our reformulation can be gauged by comparing our formal specification and verification with that of Bevier and Young. Bevier and Young state [3, page 1]

> - "We believe that our formulation provides a very clear and unambiguous characterization of the algorithm.
> - "Our machine checked proof elucidates several issues which are treated rather lightly in the published version of the proof. In particular, the invariant maintained in the recursive subcases of the algorithm is significantly more complicated than is suggested by the published proof.
>
> "The latter two advantages arise as a consequence of providing a fully formal proof, whether machine checked or not. However, the use of a powerful mechanical theorem prover as a checker is a boon in managing the complexity of the formal proof."

Regarding the first of these claims, we believe that our formulation is rather clearer and simpler (and more useful) than Bevier and Young's, but that may be a matter of taste. We believe we have demonstrated that their second claim is mistaken: our machine-checked proof is essentially the same as the published version of the proof, and we think it likely that the complexity discovered by Bevier and Young was an artifact of their formalization and of the theorem prover at their disposal. Rather than agreeing that "a powerful mechanical theorem prover ... is a boon in managing the complexity of the formal proof," we believe that a mechanical theorem prover should help the user develop reasonably clear and straightforward proofs.

In summary, we believe this small example provides some substantiation for our belief that the benefits of formal specification and verification are best assisted by rather rich specification languages that permit natural forms of expression, and by approaches to theorem proving that permit fairly direct control by the user.

Finally, recall that in the Introduction, we stated that our other motivations for undertaking this work were to see whether we derived benefits similar to those obtained in earlier formal verifications that we had undertaken (e.g, discovery of flaws in previous proofs), and to compare the difficulty of verifying of an algorithm for interactive consistency with that of one for clock synchronization.

Unlike our experience with clock synchronization algorithms, the benefits we derived from formal verification of OMIC did not include discovery of flaws in the previously published proof of correctness of the algorithm. However, building on the experience gained in the exercise described here, we have discovered an error in the algorithm (not just the proof) of Thambidurai and Park [21] for Interactive Consistency under a hybrid fault model; working with Patrick Lincoln, we have also developed and formally verified a correct algorithm for this problem.

Regarding difficulty, we can report that we found verification of OMIC to be an order of magnitude simpler than that of Interactive Convergence [15, 16]: four days work compared to about 40, and 23 subsidiary lemmas compared with nearly 200. Given its relatively small size, but rather interesting character, we invite others to try formal specification and verification of the Oral Messages Algorithm using their favorite verification system. We have used the Byzantine Generals formulation of this algorithm as one of the test cases in the development of our new Prototype Verification System, PVS [11]. Using PVS, we are now able to construct the main proofs for correctness of OMBG in under an hour. For those more interested in fault-tolerant algorithms than the performance-testing of verification systems, an interesting challenge is to develop and formally verify some of the many variants that have been proposed for the Oral Messages Algorithm. We, for example, have investigated the algorithm of Thambidurai and Park as already mentioned, and have also formally verified a generalization of the algorithm used to provide Interactive Consistency in the Draper FTP architecture [7].

Bibliography

[1] D. Basin and M. Kaufmann. The Boyer-Moore prover and Nuprl: An experimental comparison. In G. Huet and G. Plotkin, editors, *Preliminary Proceedings of the First Workshop on Logical Frameworks*, pages 43–65, Antibes, France, May 1990. Final proccedings to be published by Cambridge University Press.

[2] W. R. Bevier and W. D. Young. The design and proof of correctness of a fault-tolerant circuit. In Meyer and Schlichting [10], pages 243–260.

[3] 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.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, NY, 1988.

[5] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. Formal design and verification of a reliable computing platform for real-time control. NASA Technical Memorandum 102716, NASA Langley Research Center, Hampton, VA, October 1990.

[6] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In Meyer and Schlichting [10], pages 279–306.

[7] Jaynarayan H. Lala. A Byzantine resilient fault tolerant computer for nuclear power application. In *Fault Tolerant Computing Symposium 16*, pages 338–343, Vienna, Austria, July 1986. IEEE Computer Society.

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

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

[10] J. F. Meyer and R. D. Schlichting, editors. *Dependable Computing for Critical Applications—2*, Tucson, AZ, February 1991. Springer-Verlag, Wien, Austria, volume 6 of *Dependable Computing and Fault-Tolerant Systems*.

[11] 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, 1992. Springer Verlag, volume 607 of *Lecture Notes in Artificial Intelligence*.

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

[13] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384, July 1991.

[14] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In Vytopil [22], pages 237–257.

[15] 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.

[16] 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*, January 1993.

[17] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[18] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[19] Natarajan Shankar. Mechanical verification of a schematic Byzantine fault-tolerant clock synchronization algorithm. Technical Report SRI-CSL-91-4, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4386, July 1991.

[20] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [22], pages 217–236.

[21] 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.

[22] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, The Netherlands, January 1992. Springer Verlag, volume 571 of *Lecture Notes in Computer Science*.

[23] Jeannette Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5(4):66–76, July 1988.

[24] William D. Young. Comparing specification paradigms: Gypsy and Z. In *Proceedings 12th National Computer Security Conference*, pages 83–97, Baltimore, MD, October 1989. NBS/NCSC.

[25] 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.

Appendix A

The “Real” Specifications

In this Appendix we reproduce the “real” specifications of the algorithms employed by Bevier and Young and by ourselves. Bevier and Young’s specification differs from that of Figure 3.1 by combining the pair of mutually recursive functions into a single function with a “flag” argument; our specification is the same as that given on page 17, but is reproduced here in its raw text form.

Definition

```
(vom flg m g v l vec)
   =
(if flg
  (if (zerop m)
      (vom0 g v l vec)
      (votelist
        (pair (vom0 g v l vec)
              (vom f (sub1 m) l (vom0 g v l vec) l vec)
               l)))
  (if (listp g-list)
      (pair (vom t m (car g) (get (car g) vom0)
                 (delete (car g) l) vec)
            (vom f m (cdr g) v l vec)
            (delete (car g) l))
    (init nil (length vec))))
```

Figure A.1: Bevier and Young's Specification—The Real Version

```
OMIC: RECURSIVE function[rounds, vector, set
                       -> function[processors -> vector]] =
  (LAMBDA r, v, caucus :
     IF r = 0
       THEN (LAMBDA p :
          (LAMBDA q :
             IF member(p, caucus) AND member(q, caucus)
               THEN send(r, v(q), q, p) ELSE undef END IF))
       ELSE (LAMBDA p :
          (LAMBDA q :
             IF member(p, caucus) AND member(q, caucus)
               THEN IF p = q
                 THEN send(r, v(q), q, q)
                 ELSE maj(remove(caucus, q),
                          OMIC(r - 1, distr(r, v(q), q),
                               remove(caucus, q))(p))
                 END IF
               ELSE undef
             END IF))
       END IF)
  BY terminates
```

Figure A.2: Our Specification—The Raw Text Version

Appendix B

The Byzantine Generals Formulation of the Algorithm

We specify the Byzantine Generals formulation of the Oral Messages algorithm as a function OMBG of four arguments: G the identity of the General, m the number of rounds, t the value the General wishes to communicate, and *caucus*, the set of participants (which includes the General). OMBG will return a vector of values in which OMBG($G, m, t, caucus$)(p) is lieutenant p's opinion of the General's value. The correctness conditions are the following.

BG1_final: **Theorem** $\mathrm{ok}(p) \wedge \mathrm{ok}(q)$
$\quad\supset \mathrm{OMBG}(G, m, t, \mathrm{fullset})(p) = \mathrm{OMBG}(G, m, t, \mathrm{fullset})(q)$
BG2_final: **Theorem** $\mathrm{ok}(p) \wedge \mathrm{ok}(G) \supset \mathrm{OMBG}(G, m, t, \mathrm{fullset})(p) = t$

The specification of OMBG is rather interesting; it is due to our colleague Shankar. In the case $r = 0$, lieutenant p's component of the vector returned is simply the value received by p from the General; in the case $r > 0$, lieutenant p's component of the vector is the value the General receives from himself when $p = G$, otherwise it is the result of applying the *maj* function to the vector of values that p obtains when each of the lieutenants in the *caucus* (less G but including p himself) acts as the General in the OMBG algorithm with $r - 1$ rounds to distribute the value received by that lieutenant from the original General. Notice how the higher-order capabilities of the EHDM specification language allow us to specify the inner, iterative application of OMBG by means of a λ-abstraction, thereby avoiding the mutually recursive functions of Bevier and Young's specification.

The formal verification of OMBG is very similar to that of OMIC, and was derived from that of OMIC in less than a day. It is available from the author on request.

terminatesBG: function[processors, rounds, T, set $\rightarrow$ nat] ==
  ($\lambda\, p, r, t,$ caucus $\rightarrow$ nat : r)

OMBG: **Recursive** function[processors, rounds, T, set $\rightarrow$ vector] =
  ($\lambda\, G, r, t,$ caucus :
      if $r = 0$
        then ($\lambda\, p$: **if** caucus(p) $\wedge$ caucus(G)
          then send(r, t, G, p)
          else undef
          end if)
        else ($\lambda\, p$: **if** caucus(p) $\wedge$ caucus(G)
          then **if** $p = G$
            then send(r, t, G, G)
            else maj(caucus $- \{G\}$,
                  ($\lambda\, q$: OMBG($q, r-1,$ send(r, t, G, q), caucus $- \{G\}$)(p)))
            end if
          else undef
          end if)
        end if) **by** terminatesBG

Figure B.1: Our Formulation of the Byzantine Generals Version of the Algorithm

Appendix C

The Full Specification and Verification

The specification text and verification output that follow were processed by EHDM Version 5.2; some minor changes to the input syntax (e.g., deletion of the `PROOF` keyword) will be necessary in order to use the current version of EHDM (numbered 6.1.1).

C.1 The Specification

We reproduce here the text of our specification and verification for the IC version of the OM algorithm. The text comprises the module `consensus`. In the interests of brevity, we do not reproduce the system-generated module `consensus_tcc` that contains the "typecheck-consistency conditions" (TCCs), nor the module `top` that gives their proofs. Neither do we reproduce the library modules `noetherian`, `induction` and `functionprops`. The module `noetherian` specifies the axiom of Noetherian Induction (see, for example [15, page 99], [13, page 62] or [17, pages 57–61]), and the module `induction` (see, for example [13, page 63]) derives some more specialized induction schemes from that general formulation. One of these is used to prove the `round_induct` induction scheme over rounds that is employed here. The `functionprops` module (see, for example [15, page 99]) simply specifies an axiom of function extensionality.

C.1.1 Module "Consensus"

This module contains the specification and verification of the OMIC algorithm. Certain subsidiary concepts, such as sets and cardinality are defined here, too. Normally these concepts are imported from library modules (see, for example [13, page 66]), but so few of their properties are needed here that we have preferred to specify them in line.

---

consensus: **Module** $[m, n\colon \text{nat}]$

Exporting all

Assuming

mn_prop: **Formula** $3 * m < n$

Theory

x: **Var** nat

processors: **Type from** nat **with** $(\lambda\, x : x < n)$

rounds: **Type from** nat **with** $(\lambda\, x : x \leq m)$

T: **Type**

vector: **Type is** function[processors $\rightarrow T$]

r, s: **Var** rounds

v, v_1, v_2: **Var** vector

p, q, y, z: **Var** processors

undef: T

t: **Var** T

set: **Type is** function[processors $\rightarrow$ bool]

fullset: set == $(\lambda\, z : \text{true})$

ok: function[processors $\rightarrow$ bool]

caucus, m_1, m_2: **Var** set

$p \in m_1$: function[processors, set $\rightarrow$ bool] == $(\lambda\, p, m_1 : m_1(p))$

faulty_members: function[set $\rightarrow$ set] = $(\lambda\, m_1 : (\lambda\, z : z \in m_1 \wedge \neg \text{ok}(z)))$

$\star 1 - \{\star 2\}$: function[set, processors $\rightarrow$ set] ==
 $(\lambda\, \text{caucus}, q : \text{caucus}$ **with** $[(q) := \text{false}])$

$|\star 1|$: function[set $\rightarrow$ nat]

non_empty_ax: **Axiom** $(\exists\, p : p \in m_1) \Leftrightarrow |m_1| \neq 0$

fullset_card_ax: **Axiom** $|\text{fullset}| = n \wedge |\text{faulty\_members}(\text{fullset})| \leq m$

all_ok: **Lemma** $0 = |\text{faulty\_members}(\text{caucus})| \wedge p \in \text{caucus} \supset \text{ok}(p)$

card_remove_ax: **Axiom** $z \in m_1 \supset |m_1 - \{z\}| = |m_1| - 1$

faulty_members_card_remove_ok: **Lemma**
$z \in m_1 \wedge \text{ok}(z) \supset |\text{faulty\_members}(m_1 - \{z\})| = |\text{faulty\_members}(m_1)|$

faulty_members_card_remove_nok: **Lemma**
$z \in m_1 \wedge \neg\text{ok}(z) \supset |\text{faulty\_members}(m_1 - \{z\})| = |\text{faulty\_members}(m_1)| - 1$

maj: function[set, vector $\rightarrow T$]

majax: **Axiom** $|\text{caucus}| > 2 * |\text{faulty\_members}(\text{caucus})|$
$\wedge\ (\forall\, p : \text{ok}(p) \wedge p \in \text{caucus} \supset v(p) = t)$
$\supset \text{maj}(\text{caucus}, v) = t$

maj_ext: **Axiom** $(\forall\, p : p \in \text{caucus} \supset v_1(p) = v_2(p))$
$\supset \text{maj}(\text{caucus}, v_1) = \text{maj}(\text{caucus}, v_2)$

send: function[rounds, T, processors, processors $\rightarrow T$]

send_ax: **Axiom** $\text{ok}(p) \wedge \text{ok}(q) \supset \text{send}(r, t, q, p) = t$

distr: function[rounds, T, processors $\rightarrow$ vector] ==
$(\lambda\, r, t, p : (\lambda\, z : \text{send}(r, t, p, z)))$

terminates: function[rounds, vector, set $\rightarrow$ nat] == $(\lambda\, r, v, \text{caucus} \rightarrow \text{nat} : r)$

OMIC: **Recursive** function[rounds, vector, set $\rightarrow$ function[processors $\rightarrow$ vector]]
= ($\lambda\, r, v$, caucus :
  if $r = 0$
   then ($\lambda\, p$: ($\lambda\, q$:
      if $p \in$ caucus $\wedge\ q \in$ caucus **then** send($r, v(q), q, p$) **else** undef **end if**))
   else ($\lambda\, p$: ($\lambda\, q$:
      if $p \in$ caucus $\wedge\ q \in$ caucus
       then **if** $p = q$
        then send($r, v(q), q, q$)
        else maj(caucus $- \{q\}$,
            OMIC($r - 1$, distr($r, v(q), q$), caucus $- \{q\}$)(p))
        end if
       else undef
       end if))
   end if) **by** terminates

C1_final: **Theorem** $\text{ok}(p) \wedge \text{ok}(q)$
$\supset \text{OMIC}(m, v, \text{fullset})(p)(y) = \text{OMIC}(m, v, \text{fullset})(q)(y)$

C2_final: **Theorem** $\text{ok}(p) \wedge \text{ok}(q) \supset \text{OMIC}(m, v, \text{fullset})(p)(q) = v(q)$

C1prop: function[rounds $\rightarrow$ bool] =
$(\lambda\, r : (\forall\, p, q, y, \text{caucus}, v :$
$\text{ok}(p) \wedge \text{ok}(q) \wedge p \in \text{caucus}$
$\wedge\, q \in \text{caucus}$
$\wedge\, y \in \text{caucus} \wedge |\text{caucus}| > 3 * r \wedge r \geq |\text{faulty\_members}(\text{caucus})|$
$\supset \text{OMIC}(r, v, \text{caucus})(p)(y) = \text{OMIC}(r, v, \text{caucus})(q)(y)))$

C2prop: function[rounds $\rightarrow$ bool] =
$(\lambda\, r : (\forall\, p, q, \text{caucus}, v :$
$\text{ok}(p) \wedge \text{ok}(q) \wedge p \in \text{caucus}$
$\wedge\, q \in \text{caucus} \wedge |\text{caucus}| > 2 * |\text{faulty\_members}(\text{caucus})| + r$
$\supset \text{OMIC}(r, v, \text{caucus})(p)(q) = v(q)))$

C1: **Lemma** C1prop(r)

C2: **Lemma** C2prop(r)

Proof

Using induction, functionprops[processors, bool]

i: **Var** nat

round_prop: **Var** function[rounds $\rightarrow$ bool]

round_induct: **Lemma** (round_prop(0)
$\wedge\, (\forall\, r : r < m \wedge \text{round\_prop}(r) \supset \text{round\_prop}(r + 1)))$
$\supset \text{round\_prop}(s)$

round_induct_proof: **Prove**
round_induct $\{r \leftarrow$ **if** $i@p1$ **in** rounds **then** $i@p1$ **else** 0 **end if**$\}$ **from**
limited_induction
$\{m \leftarrow 0,$
$m_1 \leftarrow m,$
$p \leftarrow (\lambda\, i :$ **if** i **in** rounds **then** round_prop(i) **else** false **end if**),
$n \leftarrow s\}$

distr_prop: **Lemma** $\text{ok}(p) \wedge \text{ok}(q) \supset \text{distr}(r, v(p), p)(q) = v(p)$

distr_prop_proof: **Prove** distr_prop **from**
send_ax $\{t \leftarrow v(p),\ p \leftarrow q,\ q \leftarrow p\}$

OM0_prop: **Lemma** $\text{OMIC}(0, v, \text{caucus})(p)(q)$
= **if** $p \in \text{caucus} \wedge q \in \text{caucus}$ **then** $\text{send}(0, v(q), q, p)$ **else** undef **end if**

OM0_prop_proof: **Prove** OM0_prop **from** OMIC $\{r \leftarrow 0\}$

OM_prop: **Lemma** $r > 0 \supset \text{OMIC}(r, v, \text{caucus})(p)(q)$
$= $ **if** $p \in \text{caucus} \wedge q \in \text{caucus}$
then **if** $p = q$
then $\text{send}(r, v(q), q, q)$
else $\text{maj}(\text{caucus} - \{q\}, \text{OMIC}(r-1, \text{distr}(r, v(q), q), \text{caucus} - \{q\})(p))$
end if
else undef
end if

OM_prop_proof: **Prove** OM_prop **from** OMIC

OM0_ok: **Lemma** $\text{ok}(p) \wedge \text{ok}(q) \wedge p \in \text{caucus} \wedge q \in \text{caucus}$
$\supset \text{OMIC}(0, v, \text{caucus})(p)(q) = v(q)$

OM0_ok_proof: **Prove** OM0_ok **from** OM0_prop, send_ax $\{r \leftarrow 0,\ t \leftarrow v(q@c)\}$

ok_self: **Lemma** $\text{ok}(y) \wedge y \in \text{caucus} \supset \text{OMIC}(r, v_2, \text{caucus})(y)(y) = v_2(y)$

ok_self_proof: **Prove** ok_self **from**
OM_prop $\{v \leftarrow v_2,\ p \leftarrow y,\ q \leftarrow y\}$,
OM0_prop $\{v \leftarrow v_2,\ p \leftarrow y,\ q \leftarrow y\}$,
send_ax $\{p \leftarrow y,\ q \leftarrow y,\ t \leftarrow v_2(y)\}$

remove_ok_member: **Lemma**
$z \in m_1 \wedge \text{ok}(z) \supset (p \in \text{faulty\_members}(m_1 - \{z\}) \Leftrightarrow p \in \text{faulty\_members}(m_1))$

remove_ok_member_proof: **Prove** remove_ok_member **from**
faulty_members $\{z \leftarrow p\}$, faulty_members $\{m_1 \leftarrow m_1 - \{z\},\ z \leftarrow p\}$

remove_ok: **Lemma** $z \in m_1 \wedge \text{ok}(z) \supset \text{faulty\_members}(m_1 - \{z\}) = \text{faulty\_members}(m_1)$

remove_ok_proof: **Prove** remove_ok **from**
remove_ok_member $\{p \leftarrow a@p2\}$,
extensionality $\{F \leftarrow \text{faulty\_members}(m_1),\ G \leftarrow \text{faulty\_members}(m_1 - \{z\})\}$

remove_nok_member: **Lemma**
$z \in m_1 \wedge \neg\text{ok}(z) \supset (p \in \text{faulty\_members}(m_1 - \{z\}) \Leftrightarrow p \in \text{faulty\_members}(m_1) - \{z\})$

remove_nok_member_proof: **Prove** remove_nok_member **from**
faulty_members $\{z \leftarrow p\}$, faulty_members $\{m_1 \leftarrow m_1 - \{z\},\ z \leftarrow p\}$

remove_nok: **Lemma** $z \in m_1 \wedge \neg\text{ok}(z)$
$\supset \text{faulty\_members}(m_1 - \{z\}) = \text{faulty\_members}(m_1) - \{z\}$

remove_nok_proof: **Prove** remove_nok **from**
remove_nok_member $\{p \leftarrow a@p2\}$,
extensionality $\{F \leftarrow \text{faulty\_members}(m_1) - \{z\},\ G \leftarrow \text{faulty\_members}(m_1 - \{z\})\}$

faulty_members_card_remove_ok_proof: **Prove** faulty_members_card_remove_ok **from**
remove_ok

faulty_members_card_remove_nok_proof: **Prove** faulty_members_card_remove_nok **from**
remove_nok, faulty_members, card_remove_ax $\{m_1 \leftarrow \text{faulty\_members}(m_1@c)\}$

ok_card_remove: **Lemma**
$r < m \wedge q \in \text{caucus} \wedge \text{ok}(q)$
$\supset |\text{caucus}| > 2 * |\text{faulty\_members}(\text{caucus})| + r + 1$
$\supset |\text{caucus} - \{q\}| > 2 * |\text{faulty\_members}(\text{caucus} - \{q\})| + r$

ok_card_remove_proof: **Prove** ok_card_remove **from**
card_remove_ax $\{m_1 \leftarrow \text{caucus},\ z \leftarrow q\}$,
faulty_members_card_remove_ok $\{m_1 \leftarrow \text{caucus},\ z \leftarrow q\}$

ok_others: **Lemma** $r < m$
$\wedge\ |\text{caucus} - \{q\}| > 2 * |\text{faulty\_members}(\text{caucus} - \{q\})|$
$\wedge\ \text{ok}(y) \wedge \text{ok}(q)$
$\wedge\ y \in \text{caucus}$
$\wedge\ q \in \text{caucus}$
$\wedge\ y \neq q$
$\wedge\ (\forall\, z, v_1 :$
$z \in \text{caucus} \wedge \text{ok}(z) \wedge z \neq q$
$\supset \text{OMIC}(r, v_1, \text{caucus} - \{q\})(y)(z) = v_1(z))$
$\supset \text{OMIC}(r + 1, v_2, \text{caucus})(y)(q) = v_2(q)$

next_round: function[rounds $\rightarrow$ rounds] ==
$(\,\lambda\, r \rightarrow$ rounds : **if** $r < m$ **then** $r + 1$ **else** 0 **end if**$)$

ok_others_proof: **Prove**
ok_others $\{z \leftarrow p@p1,\ v_1 \leftarrow \text{distr}(\text{next\_round}(r), v_2(q), q)\}$ **from**
majax
$\{\text{caucus} \leftarrow \text{caucus} - \{q\}$,
$v \leftarrow \text{OMIC}(r, \text{distr}(\text{next\_round}(r), v_2(q), q), \text{caucus} - \{q\})(y)$,
$t \leftarrow v_2(q)\}$,
OM_prop $\{r \leftarrow \text{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow y\}$,
distr_prop $\{r \leftarrow \text{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow q,\ q \leftarrow y\}$,
distr_prop $\{r \leftarrow \text{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow q,\ q \leftarrow q\}$,
distr_prop $\{r \leftarrow \text{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow q,\ q \leftarrow p@p1\}$

C2prop_0: **Lemma** C2prop(0)

C2prop_0_proof: **Prove** C2prop_0 **from**
C2prop $\{r \leftarrow 0\}$,
OM0_ok $\{p \leftarrow p@p1,\ q \leftarrow q@p1,\ v \leftarrow v@p1,\ \text{caucus} \leftarrow \text{caucus}@p1\}$

C2prop_r: **Lemma** $r < m \wedge \text{C2prop}(r) \supset \text{C2prop}(r + 1)$

remove_others: **Lemma** $p \in \text{caucus} \wedge p \neq q \supset p \in \text{caucus} - \{q\}$

remove_others_proof: **Prove** remove_others

C2prop_r_proof: **Prove** C2prop_r **from**
C2prop
$\{v \leftarrow v_1@P3,$
$q \leftarrow z@p3,$
$p \leftarrow p@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2 - \{q@p2\}\},$
C2prop $\{r \leftarrow \text{next\_round}(r)\},$
ok_others
$\{q \leftarrow q@p2,$
$y \leftarrow p@p2,$
$v_2 \leftarrow v@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2\},$
ok_self
$\{r \leftarrow \text{next\_round}(r),$
$y \leftarrow p@p2,$
$v_2 \leftarrow v@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2\},$
ok_card_remove $\{\text{caucus} \leftarrow \text{caucus}@p2,\ q \leftarrow q@p2\},$
remove_others $\{\text{caucus} \leftarrow \text{caucus}@p2,\ q \leftarrow q@p2,\ p \leftarrow p@p2\},$
remove_others $\{\text{caucus} \leftarrow \text{caucus}@p2,\ q \leftarrow q@p2,\ p \leftarrow z@p3\}$

C2_proof: **Prove** C2 **from**
round_induct $\{\text{round\_prop} \leftarrow \text{C2prop},\ s \leftarrow r\},$
C2prop_0,
C2prop_r $\{r \leftarrow r@p1\}$

agree_nok: **Lemma** $r < m$

$$
\begin{aligned}
&\wedge |\text{caucus}| > 3 * (r+1)\\
&\quad \wedge r + 1 \geq |\text{faulty\_members}(\text{caucus})|\\
&\quad\quad \wedge \text{ok}(p) \wedge \text{ok}(q)\\
&\quad\quad\quad \wedge p \in \text{caucus}\\
&\quad\quad\quad\quad \wedge q \in \text{caucus}\\
&\quad\quad\quad\quad\quad \wedge y \in \text{caucus}\\
&\quad\quad\quad\quad\quad\quad \wedge \neg\text{ok}(y)\\
&\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\, z, v_1 :\\
&\quad\quad\quad\quad\quad\quad\quad\quad z \in \text{caucus} - \{y\}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \supset \text{OMIC}(r, v_1, \text{caucus} - \{y\})(p)(z)\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad = \text{OMIC}(r, v_1, \text{caucus} - \{y\})(q)(z))\\
&\supset \text{OMIC}(r+1, v_2, \text{caucus})(p)(y) = \text{OMIC}(r+1, v_2, \text{caucus})(q)(y)
\end{aligned}
$$

agree_nok_proof: **Prove**
  agree_nok $\{z \leftarrow p@p3,\ v_1 \leftarrow \mathrm{distr}(\mathrm{next\_round}(r), v_2(y), y)\}$ **from**
  OM_prop $\{r \leftarrow \mathrm{next\_round}(r),\ v \leftarrow v_2,\ q \leftarrow y\}$,
  OM_prop $\{r \leftarrow \mathrm{next\_round}(r),\ v \leftarrow v_2,\ q \leftarrow y,\ p \leftarrow q\}$,
  maj_ext
    $\{\mathrm{caucus} \leftarrow \mathrm{caucus} - \{y\}$,
    $v_1 \leftarrow \mathrm{OMIC}(r, \mathrm{distr}(\mathrm{next\_round}(r), v_2(y), y), \mathrm{caucus} - \{y\})(p)$,
    $v_2 \leftarrow \mathrm{OMIC}(r, \mathrm{distr}(\mathrm{next\_round}(r), v_2(y), y), \mathrm{caucus} - \{y\})(q)\}$,
  distr_prop $\{r \leftarrow \mathrm{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow y\}$,
  distr_prop $\{r \leftarrow \mathrm{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow y,\ q \leftarrow y\}$,
  distr_prop $\{r \leftarrow \mathrm{next\_round}(r),\ v \leftarrow v_2,\ p \leftarrow y,\ q \leftarrow p@p1\}$

agree_ok: **Lemma** $r < m$
  $\wedge\ |\mathrm{caucus}| > 3 * (r+1)$
  $\wedge\ r + 1 \geq |\mathrm{faulty\_members}(\mathrm{caucus})|$
  $\wedge\ \mathrm{ok}(p) \wedge \mathrm{ok}(q) \wedge p \in \mathrm{caucus} \wedge q \in \mathrm{caucus} \wedge y \in \mathrm{caucus} \wedge \mathrm{ok}(y)$
  $\supset \mathrm{OMIC}(r+1, v_2, \mathrm{caucus})(p)(y) = \mathrm{OMIC}(r+1, v_2, \mathrm{caucus})(q)(y)$

agree_ok_proof: **Prove** agree_ok **from**
  C2 $\{r \leftarrow \mathrm{next\_round}(r)\}$,
  C2prop $\{r \leftarrow \mathrm{next\_round}(r),\ q \leftarrow y,\ v \leftarrow v_2\}$,
  C2prop $\{r \leftarrow \mathrm{next\_round}(r),\ p \leftarrow q,\ q \leftarrow y,\ v \leftarrow v_2\}$

all_ok_proof: **Prove** all_ok **from**
  non_empty_ax $\{m_1 \leftarrow \mathrm{faulty\_members}(\mathrm{caucus})\}$,
  faulty_members $\{m_1 \leftarrow \mathrm{caucus},\ z \leftarrow p\}$

C1prop_0: **Lemma** C1prop(0)

C1prop_0_proof: **Prove** C1prop_0 **from**
  C1prop $\{r \leftarrow 0\}$,
  OM0_ok $\{p \leftarrow p@p1,\ q \leftarrow y@p1,\ v \leftarrow v@p1,\ \mathrm{caucus} \leftarrow \mathrm{caucus}@p1\}$,
  OM0_ok $\{p \leftarrow q@p1,\ q \leftarrow y@p1,\ v \leftarrow v@p1,\ \mathrm{caucus} \leftarrow \mathrm{caucus}@p1\}$,
  all_ok $\{p \leftarrow y@p1,\ \mathrm{caucus} \leftarrow \mathrm{caucus}@p1\}$,
  nat_invariant $\{\mathrm{nat\_var} \leftarrow |\mathrm{faulty\_members}(\mathrm{caucus}@p1)|\}$

C1prop_r: **Lemma** $r < m \wedge \mathrm{C1prop}(r) \supset \mathrm{C1prop}(r+1)$

C1prop_r_proof: **Prove** C1prop_r **from**
C1prop
$\{v \leftarrow v_1@p3,$
$y \leftarrow z@p3,$
$p \leftarrow p@p2,$
$q \leftarrow q@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2 - \{y@p2\}\},$
C1prop $\{r \leftarrow \text{next\_round}(r)\}$,
agree_nok
$\{v_2 \leftarrow v@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2,$
$p \leftarrow p@p2,$
$q \leftarrow q@p2,$
$y \leftarrow y@p2\},$
agree_ok
$\{v_2 \leftarrow v@p2,$
$\text{caucus} \leftarrow \text{caucus}@p2,$
$p \leftarrow p@p2,$
$q \leftarrow q@p2,$
$y \leftarrow y@p2\},$
remove_others $\{p \leftarrow p@p2,\ q \leftarrow y@p2,\ \text{caucus} \leftarrow \text{caucus}@p2\}$,
remove_others $\{p \leftarrow q@p2,\ q \leftarrow y@p2,\ \text{caucus} \leftarrow \text{caucus}@p2\}$,
card_remove_ax $\{m_1 \leftarrow \text{caucus}@p2,\ z \leftarrow y@p2\}$,
faulty_members_card_remove_nok $\{m_1 \leftarrow \text{caucus}@p2,\ z \leftarrow y@p2\}$

C1_proof: **Prove** C1 **from**
round_induct $\{\text{round\_prop} \leftarrow \text{C1prop},\ s \leftarrow r\}$,
C1prop_0,
C1prop_r $\{r \leftarrow r@p1\}$

C1_final_proof: **Prove** C1_final **from**
C1 $\{r \leftarrow m\}$, C1prop $\{r \leftarrow m,\ \text{caucus} \leftarrow \text{fullset}\}$, fullset_card_ax, mn_prop

C2_final_proof: **Prove** C2_final **from**
C2 $\{r \leftarrow m\}$, C2prop $\{r \leftarrow m,\ \text{caucus} \leftarrow \text{fullset}\}$, fullset_card_ax, mn_prop

End consensus

C.2 Proof-Chain Analysis

The following pages reproduce the output from the EHDM proof-chain analyzer in "terse mode" applied to the formula `C1_final` in module `consensus`. The analysis for `C2_final` is similar. The EHDM proof-chain analyzer examines the macroscopic structure of a verification—checking that all the premises used in a proof are either axioms, definitions, or formulas which are, themselves, the target of a successful proof elsewhere in the verification. If any formulas are used from a module having an `assuming` clause, then the proof-chain analyzer checks that those assumptions are discharged by successful proofs; similarly, if formulas are used from a module having a TCC module, then the proof-chain analyzer checks that all the TCCs in that module are discharged by successful proofs. The proof-chain analyzer ignores unsuccessful proofs (such as automatically-generated TCC proofs) when a successful proof for the same formula can be found. The "terse mode" output reproduced here provides a commentary on only the "interesting" cases, namely proof obligations involving assuming clauses and TCCs, and a summary. All the proofs listed in the summary were performed by the EHDM theorem prover in "checking mode."

```
Terse proof chain for formula C1_final in module consensus
Interesting cases from the analysis follow; see summary for status

Use of the formula
  consensus[EXPR, EXPR].C1_final
requires the following TCCs to be proven
  consensus_tcc[EXPR, EXPR].processors_TCC1
  consensus_tcc[EXPR, EXPR].rounds_TCC1
  consensus_tcc[EXPR, EXPR].OM_TCC1
  consensus_tcc[EXPR, EXPR].OM_TCC2
  consensus_tcc[EXPR, EXPR].C1_final_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_TCC2
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC2
  consensus_tcc[EXPR, EXPR].OM0_prop_TCC1
  consensus_tcc[EXPR, EXPR].OM_prop_TCC1
  consensus_tcc[EXPR, EXPR].OM0_ok_TCC1
  consensus_tcc[EXPR, EXPR].ok_others_TCC1
  consensus_tcc[EXPR, EXPR].next_round_TCC1
  consensus_tcc[EXPR, EXPR].C2prop_r_TCC1
  consensus_tcc[EXPR, EXPR].agree_nok_TCC1
  consensus_tcc[EXPR, EXPR].agree_ok_TCC1
  consensus_tcc[EXPR, EXPR].C1prop_r_TCC1
  consensus_tcc[EXPR, EXPR].C1_final_proof_TCC1

Use of the formula
```

```
  induction.limited_induction
requires the following TCCs to be proven
  induction_tcc.ind_m_proof_TCC1

Use of the formula
  noetherian[naturalnumber, induction.prev].general_induction
requires the following assumptions to be discharged
  noetherian[naturalnumber, induction.prev].well_founded

   ================== SUMMARY ==================

The proof chain is complete

The axioms and assumptions at the base are:
  consensus[EXPR, EXPR].card_remove_ax
  consensus[EXPR, EXPR].fullset_card_ax
  consensus[EXPR, EXPR].maj_ext
  consensus[EXPR, EXPR].majax
  consensus[EXPR, EXPR].mn_prop
  consensus[EXPR, EXPR].non_empty_ax
  consensus[EXPR, EXPR].send_ax
  functionprops[EXPR, EXPR].extensionality
  noetherian[EXPR, EXPR].general_induction
Total: 9

The definitions and type-constraints are:
  consensus[EXPR, EXPR].C1prop
  consensus[EXPR, EXPR].C2prop
  consensus[EXPR, EXPR].OM
  consensus[EXPR, EXPR].faulty_members
  naturalnumbers.nat_invariant
Total: 5

The formulae used are:
  consensus[EXPR, EXPR].C1
  consensus[EXPR, EXPR].C1_final
  consensus[EXPR, EXPR].C1prop_0
  consensus[EXPR, EXPR].C1prop_r
  consensus[EXPR, EXPR].C2
  consensus[EXPR, EXPR].C2prop_0
  consensus[EXPR, EXPR].C2prop_r
  consensus[EXPR, EXPR].OM0_ok
  consensus[EXPR, EXPR].OM0_prop
  consensus[EXPR, EXPR].OM_prop
  consensus[EXPR, EXPR].agree_nok
  consensus[EXPR, EXPR].agree_ok
```

```
  consensus[EXPR, EXPR].all_ok
  consensus[EXPR, EXPR].distr_prop
  consensus[EXPR, EXPR].faulty_members_card_remove_nok
  consensus[EXPR, EXPR].faulty_members_card_remove_ok
  consensus[EXPR, EXPR].ok_card_remove
  consensus[EXPR, EXPR].ok_others
  consensus[EXPR, EXPR].ok_self
  consensus[EXPR, EXPR].remove_nok
  consensus[EXPR, EXPR].remove_nok_member
  consensus[EXPR, EXPR].remove_ok
  consensus[EXPR, EXPR].remove_ok_member
  consensus[EXPR, EXPR].remove_others
  consensus[EXPR, EXPR].round_induct
  consensus_tcc[EXPR, EXPR].C1_final_TCC1
  consensus_tcc[EXPR, EXPR].C1_final_proof_TCC1
  consensus_tcc[EXPR, EXPR].C1prop_r_TCC1
  consensus_tcc[EXPR, EXPR].C2prop_r_TCC1
  consensus_tcc[EXPR, EXPR].OM0_ok_TCC1
  consensus_tcc[EXPR, EXPR].OM0_prop_TCC1
  consensus_tcc[EXPR, EXPR].OM_TCC1
  consensus_tcc[EXPR, EXPR].OM_TCC2
  consensus_tcc[EXPR, EXPR].OM_prop_TCC1
  consensus_tcc[EXPR, EXPR].agree_nok_TCC1
  consensus_tcc[EXPR, EXPR].agree_ok_TCC1
  consensus_tcc[EXPR, EXPR].next_round_TCC1
  consensus_tcc[EXPR, EXPR].ok_others_TCC1
  consensus_tcc[EXPR, EXPR].processors_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_TCC2
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC1
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC2
  consensus_tcc[EXPR, EXPR].rounds_TCC1
  induction.basic_induction
  induction.induction_m
  induction.limited_induction
  induction_tcc.ind_m_proof_TCC1
  noetherian[naturalnumber, induction.prev].well_founded
Total: 49

The completed proofs are:
  consensus[EXPR, EXPR].C1_final_proof
  consensus[EXPR, EXPR].C1_proof
  consensus[EXPR, EXPR].C1prop_0_proof
  consensus[EXPR, EXPR].C1prop_r_proof
  consensus[EXPR, EXPR].C2_proof
  consensus[EXPR, EXPR].C2prop_0_proof
```

```
  consensus[EXPR, EXPR].C2prop_r_proof
  consensus[EXPR, EXPR].OM0_ok_proof
  consensus[EXPR, EXPR].OM0_prop_proof
  consensus[EXPR, EXPR].OM_prop_proof
  consensus[EXPR, EXPR].agree_nok_proof
  consensus[EXPR, EXPR].agree_ok_proof
  consensus[EXPR, EXPR].all_ok_proof
  consensus[EXPR, EXPR].distr_prop_proof
  consensus[EXPR, EXPR].faulty_members_card_remove_nok_proof
  consensus[EXPR, EXPR].faulty_members_card_remove_ok_proof
  consensus[EXPR, EXPR].ok_card_remove_proof
  consensus[EXPR, EXPR].ok_others_proof
  consensus[EXPR, EXPR].ok_self_proof
  consensus[EXPR, EXPR].remove_nok_member_proof
  consensus[EXPR, EXPR].remove_nok_proof
  consensus[EXPR, EXPR].remove_ok_member_proof
  consensus[EXPR, EXPR].remove_ok_proof
  consensus[EXPR, EXPR].remove_others_proof
  consensus[EXPR, EXPR].round_induct_proof
  consensus_tcc[EXPR, EXPR].C1_final_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].C1_final_proof_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].C1prop_r_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].C2prop_r_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].OM0_ok_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].OM0_prop_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].OM_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].OM_TCC2_PROOF
  consensus_tcc[EXPR, EXPR].OM_prop_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].agree_nok_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].agree_ok_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].next_round_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].ok_others_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].round_induct_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].round_induct_TCC2_PROOF
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC1_PROOF
  consensus_tcc[EXPR, EXPR].round_induct_proof_TCC2_PROOF
  induction.discharge
  induction.ind_m_proof
  induction.ind_proof
  induction.limited_proof
  induction_tcc.ind_m_proof_TCC1_PROOF
  top[EXPR, EXPR].processors_TCC1_PROOF
  top[EXPR, EXPR].rounds_TCC1_PROOF
Total: 49
```