NASA Contractor Report 182099

Machine Checked Proofs of the Design and Implementation of a Fault-Tolerant Circuit

William R. Bevier
William D. Young

Computational Logic, Inc. Austin, Texas

Contract NAS1-18878
November 1990

Abstract

We describe a formally verified implementation of the ‘‘Oral Messages’’ algorithm of Pease, Shostak, and Lamport [7, 8]. An abstract implementation of the algorithm is verified to achieve interactive consistency in the presence of faults. This abstract characterization is then mapped down to a hardware level implementation which inherits the fault-tolerant characteristics of the abstract version. All steps in the proof were checked with the Boyer-Moore theorem prover. A significant result of this work is the demonstration of a fault-tolerant device that is formally specified and whose implementation is proved correct with respect to this specification. A significant simplifying assumption is that the redundant processors behave synchronously. We also describe a mechanically checked proof that the Oral Messages algorithm is ‘‘optimal’’ in the sense that no algorithm which achieves agreement via similar message passing can tolerate a larger proportion of faulty processors.

Key words. Fault tolerance, mechanical theorem proving, program verification, specification.

Acknowledgement

This work was sponsored in part at Computational Logic, Inc. by National Aeronautics and Space Administration Langley Research Center (NAS1-18878). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., NASA Langley Research Center or the U.S. Government.

1. Introduction

A key problem facing the designers of systems which attempt to ensure fault tolerance by redundant processing is how to guarantee that the processors reach agreement, even when one or more processing units are faulty. This problem, called the Byzantine Generals problem or the problem of achieving *interactive consistency*, was described and solved in certain cases by Pease, Shostak, and Lamport [7, 8]. They provided an extremely clever algorithm called the ‘‘Oral Messages’’ (*OM*) algorithm which implements a solution to this problem. They also proved that under certain assumptions about the type of interprocess communication, the problem is solvable if and only if the total number of processors exceeds three times the number of faulty processors.

We have performed a machine checked proof using the Boyer-Moore theorem prover [1, 3] that an ‘‘abstract implementation’’ of the OM Algorithm does achieve interactive consistency in the presence of faults. Mechanical checking of this proof is significant for several reasons.

- It is the first machine checked proof of which we are aware of this quite difficult algorithm.
- We believe that our formalization 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 consequences of providing a fully formal proof, whether machine checked or not. However, the use of a powerful mechanical theorem prover as a proof checker is a boon in managing the complexity of the formal proof.

We have also verified a hardware implementation of *OM*(1)—the instance of *OM* which tolerates one faulty process when there are at least three non-faulty processes—as part of the implementation of a fault-tolerant device. Our approach to achieving this was to use our verified ‘‘abstract implementation’’ as a design specification. We then defined a hardware-level characterization of the algorithm and proved that our low-level version is a correct implementation of the high-level version. As a consequence of this proof, we are guaranteed that our low-level implementation achieves interactive consistency. This verified low-level description has been physically realized in programmable logic arrays. A significant assumption in our design is that the redundant processors behave synchronously. Future work will be directed at eliminating the need for this requirement.

Finally, we have machine checked the proof [8] that no algorithm exists which achieves interactive consistency via an exchange of ‘‘oral’’ messages if the number of faulty processors is at least one third of the total. This theorem shows that the *OM* algorithm is ‘‘optimal’’ in the sense that no algorithm which achieves interactive consistency purely via message exchange can tolerate a larger proportion of faulty processors. This portion of the work was primarily an exercise in formal specification and mechanical theorem proving. Attempting to specify and prove this theorem within our chosen formal framework of the Boyer-Moore logic is an interesting challenge for several reasons.

- It has typically been quite difficult to prove a negative existential statement in the Boyer-Moore logic, except in cases where the range of the quantifier is inductively defined, which is not true here.
- Statement of the theorem requires consideration of some second order concepts. The Boyer-Moore logic is first order.

We believe that the solutions to these problems adopted here indicate a rather surprising range of expressive power of the Boyer-Moore logic and the benefits provided by several recent enhancements to the logic and theorem prover.

The paper is organized as follows. The following section describes our formal specification of the Oral Messages algorithm and its correctness properties. Section 3.4 describes several steps massaging the specification to make it amenable for mapping to an implementation. In section 4.1 we describe our implementation of the algorithm; section 4.2 sketches the proof that the implementation is correct. In section 5 we describe our formalization and proof of the theorem that no algorithm achieves interactive consistency via exchange of oral messages if the number of faulty processors is at least a third of the total. Finally, section 6 gives some of our conclusions and observations on the significance of this work.

2. Interactive Consistency and the Oral Messages Algorithm

2.1 Interactive Consistency

The problem addressed by the interactive consistency algorithm is the following: given a number of communicating processors, how can they arrive at a consistent common view of the system if there are faulty processors among them which potentially send conflicting information to different parts of the system. Lamport, Shostak, and Pease [7] describe the problem in terms of the rather colorful metaphor of Byzantine Generals attempting to arrive at a common battle plan through an exchange of messages. One or more of the generals may be traitorous and attempt to thwart the loyal generals by preventing them from reaching agreement.

Pease, Shostak, and Lamport phrase the problem in terms of a single commanding general communicating with a number of lieutenant generals. In this case we desire an algorithm which guarantees the following.

A commanding general must send an order to his n - 1 lieutenant generals such that

IC1. All loyal lieutenants obey the same order.

IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Conditions IC1 and IC2 are called the *interactive consistency* conditions. [7]

We assume that the generals communicate only via *oral messages*. That is, their communication is assumed to have the following characteristics.

1. Every message that is sent is correctly delivered.
2. The receiver of a message knows who sent it.
3. The absence of a message can be detected.

In practice, we desired to use some interactive consistency algorithm in the design of a fault-tolerant device. Sensor values are read individually by a distributed collection of replicated processors (generals). These values are exchanged (as oral messages) among the processors in such a way that all non-faulty processors achieve a consistent common view of the state of the system. On the basis of this common view each non-faulty processor produces a signal to an external actuator associated with that processor. By the interactive consistency conditions, each non-faulty processor should have the same information and hence produce the same actuator value. Thus, even in the presence of faulty processors, non-faulty processors would produce identical actuator values.

2.2 Review of the Algorithm

The ‘‘Oral Messages’’ algorithm OM(m) is inductively defined for all nonnegative integers m, and describes the communication of an order by the commander to each of n - 1 lieutenants. The description of the algorithm from [7] is quoted in Figure 1. The execution of the algorithm for four processors where exactly one is faulty is illustrated in Figures 2 and 3.

Figure 2 illustrates the situation in which a loyal commanding general g sends its private value v to each of three lieutenants, one of whom is traitorous. The traitorous lieutenant (p_1 in this case) sends arbitrary values (or no value at all) to the other lieutenants. However, each of the loyal lieutenants receives one reliable value from the commanding general and another relayed by the other loyal lieutenant. Taking the majority of the three values received, the loyal lieutenants arrive at a consistent common view of the general’s value.

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.

(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), m > 0.

(1) The commander sends his value to every lieutenant. For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM(m - 1) to send the value v_i to each of the n - 2 other lieutenants.

(2) 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 OM(m - 1)), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*(v_1,...,v_{n-1}).

Figure 1: The Oral Messages Algorithm—The Journal Version

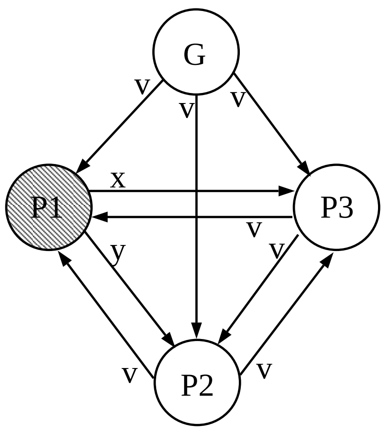

Figure 2: Four Communicating Processors with One Faulty Lieutenant

Figure 3 illustrates one situation that may occur if the commanding general is faulty. The lieutenants each receive different values which they then faithfully relay to their comrades. Since no lieutenant receives a majority, each records the default value of RETREAT. Again, IC1 and IC2 are satisfied.

These two scenarios illustrate how the three lieutenants can arrive at a consistent view of the general's

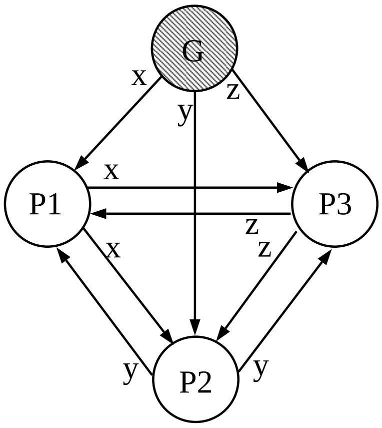

Figure 3: Four Communicating Processors with a Faulty Commander

value via this single exchange of messages provided that there is no more than one traitor among them. In general, with *n* processors of which *m* are faulty the OM algorithm achieves interactive consistency with *m* exchanges if $n \geq 3m + 1$. In section 5 we discuss the proof [8] that there is no scheme by which a group of processes can reliably reach agreement if a larger proportion are faulty.

3. The Specification

3.1 Our Formal Definition of the Algorithm

We have formalized a version of the Oral Messages algorithm in the computational logic of Boyer and Moore [1, 3]. An interesting aspect of this formalization is that, except for a few simple subsidiary definitions, the entire complexity of the algorithm is captured in around 15 lines of ‘‘code.’’ We introduce these subsidiary definitions and then explain the formal version of the algorithm itself.

The basic data structure maintained by our formalization of the algorithm is a *vector* of sequences of messages; the vector is indexed by process names. We assume for convenience that process names are simply numbers in the range **`[0..n-1]`**. Figure 4 introduces some subsidiary functions we need to describe the Oral Messages algorithm and its correctness properties. All expressions are in the Lisp-like prefix notation of the Boyer-Moore logic. (See [3] for a complete description of the syntax of our notation.)

To formalize the notion of interprocess communication, we also introduce the function **`send`**. The

`(delete x l)`	returns a list identical to **`l`** with the first occurrence (if any) of **`x`** removed.
`(indexlist n)`	returns the list of numbers **`(0 1 ... n)`**.
`(init v n)`	creates a vector of length **`n`**, all of whose elements have value **`v`**.
`(get i vec)`	fetches the **`i`**th element (zero based) from **`vec`**.
`(length lst)`	returns the number of (top-level) elements in list **`lst`**.
`(majority lst)`	returns the majority element of **`lst`**, if one exists; otherwise, returns some fixed token.
`(pair a b lst)`	for vectors **`a`** and **`b`** and list **`lst`** of process names, replace in **`b`** the value of each process in **`lst`** with the pair consisting of its value in **`a`** and its (old) value in **`b`**. That is, if **`i`** ∈ **`l`**, then **`(get i (pair a b l))`** is **`(cons (get i a) (get i b))`**.
`(put i v vec)`	replaces the **`i`**th element of **`vec`** with **`v`**.
`(select i lst)`	returns the list of every **`i`**th element of **`lst`**.
`(tablep n lst)`	returns **`T`** or **`F`** depending upon whether **`lst`** is a list of **`n`**-tuples.
`(votelist lst)`	returns the list obtained by applying **`majority`** to each of the elements of **`lst`**.

Figure 4: Some Elementary Functions

expression **`(send v i j)`** denotes the sending by process **`i`** of value **`v`** to process **`j`**.[1]

We are now ready to describe the algorithm itself. This is formally characterized in the three functions **`vom0`**, **`vom*`**, and **`voml*`** displayed in Figure 5. Function **`vom0`** implements the step in the algorithm in which the general distributes its value to each of the lieutenants. The function takes as arguments the general's name **`g`** and value **`v`**, a list **`l`** of lieutenants, and a vector in which to record these sends. The result is an updated vector in which each lieutenant **`i`** on list **`l`** is bound to **`(send v g i)`**, i.e., the slot in the vector indexed by **`i`** contains that value. This represents the first round of communication in which the general broadcasts his value to all lieutenants. Notice that the initial value of **`vec`** is irrelevant if **`l`** contains all indices but **`g`**.

The two functions **`vom*`** and **`voml*`** are mutually recursive functions which accomplish *m* rounds of message exchange among the lieutenants. Conceptually, **`vom*`** is the top-level function which takes as arguments the number **`m`** of rounds of communication, the general's name and value, a list **`l`** of lieutenant names, and the vector in which the message traffic is recorded. It returns a vector in which each

[1]More accurately, it denotes process **`j`**'s *report* of the value that process **`i`** sent to it. If both the sending and receiving processes are non-faulty, **`(send v i j)`** should reduce to **`v`**. We rely on this fact in the proof.

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 l 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 5: The Oral Messages Algorithm—Mutually Recursive Version

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 his value directly (the call to **vom0**), and voting on each element in the resulting map (the call to **votelist**).

The function **voml*** is the second of the pair of the mutually recursive functions which implement the exchange of messages. It 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' 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.

To get a feel for how the algorithm works, consider again the scenario illustrated in Figure 2 in which a loyal general **`g`** distributes value **`v`** to three lieutenants of which at most one is traitorous. Assume that the general is process 0, and the other processes are numbered 1, 2, and 3.[2] This is described formally by the call **`(vom* 1 g v (list  p1 p2 p3) vec)`**, where **`vec`** is a vector of 4 **`nil`**'s. The first parameter specifies both the number of rounds of message exchange and the maximum number of faulty processors. Expanding the definition of the function **`vom*`** we obtain:

```
(vom* 1 g v (list p1 p2 p3) vec)
   =
(votelist (pair (vom0 g v (list p1 p2 p3) vec)
                (voml* 0
                       (list p1 p2 p3)
                       (vom0 g v (list p1 p2 p3) vec)
                       (list p1 p2 p3) vec)
                (list p1 p2 p3)))
```

Notice that this involves two calls to **`(vom0 g v (list p1 p2 p3)  vec)`**, our representation of the step in the algorithm described informally as ''the commander sends his value to every lieutenant.'' This call to **`vom0`** returns a vector which records that each lieutenant has received a single message from the commander. This is represented as the vector labeled *vom0-result* below:

```
[nil,
 (send v g p1),
 (send v g p2),
 (send v g p3)]
```

vom0-result

Our expansion of the call to **`vom*`** also involves a ''recursive'' call to our other function **`voml*`**. The intended semantics of **`voml*`** is that each of the lieutenants should take the values received in the previous step and distribute them to the other lieutenants. The function **`voml*`** iterates down the structure of the list returned from **`vom0`** and sends the value received from the commander by each of the lieutenants on to each of the *other* lieutenants (we are careful not to send the value from any lieutenant to itself). The result is the vector *voml*-result* below:

```
[<>,
 <(send (send v g p2) p2 p1),
  (send (send v g p3) p3 p1)>,
 <(send (send v g p1) p1 p2),
  (send (send v g p3) p3 p2)>,
 <(send (send v g p1) p1 p3),
  (send (send v g p2) p2 p3)>]
```

voml-result*

This signifies, for example, that **`p1`** has received two message: one containing the value that **`p2`** said it

[2]We'll keep the names **`g`**, **`p1`**, **`p2`**, and **`p3`** to make the example easier to follow.

obtained from **g** and another containing the value that **p3** said it received from **g**.

We can see now that:

```
(vom* 1 g v (list p1 p2 p3))
  =
(votelist (pair vom0-result voml*-result (list p1 p2 p3)))
```

We need the application of **pair** here because *voml*-result* records the results of the exchange of messages but not the initial ''broadcast'' from the commander. Expanding **pair** we obtain:

```
(votelist
    [<>,
     <(send v g p1),
      (send (send v g p2) p2 p1),
      (send (send v g p3) p3 p1)>,
     <(send v g p2),
      (send (send v g p1) p1 p2),
      (send (send v g p3) p3 p2)>,
     <(send v g p3),
      (send (send v g p1) p1 p3),
      (send (send v g p2) p2 p3)>])
```

Thus **p2**, say, computes its view of **g**'s value by taking the majority element of the value received directly from **g** itself, the value that **p1** said it received from **g**, and the value that **p3** said that it received from **g**. Notice that if we assume that **p1** is the only faulty process, then **p2** receives at least two reliable values. Consider the messages received by **p3** to convince yourself that **p2** and **p3** must come to the same conclusion about **g**'s value.[3]

An alternative way of looking at this final computation is by applying our earlier observation that if both $\mathtt{p_i}$ and $\mathtt{p_j}$ are non-faulty, then $\mathtt{(send\ v\ p_i\ p_j)} = \mathtt{v}$. Again assuming that **p1** is the only faulty process, the vector element for **p2**, for example,

```
<(send v g p2),
 (send (send v g p1) p1 p2),
 (send (send v g p3) p3 p2)>,
```

reduces to

```
<v, (send (send v g p1) p1 p2), v>.
```

Applying the **majority** function to this list clearly yields **v**.

One aspect of the discussion above is slightly misleading. Our functions **vom*** and **voml*** are presented as mutually recursive; these would not be acceptable to the Boyer-Moore *definition principle*. However, it

[3]Since **p1** is faulty, we do not care what value it obtains.

is easy to turn these mutually recursive definitions into a single function definition which is acceptable by a trick well known to Boyer-Moore users. The actual definition is given in Figure 6. The **flg** parameter is a boolean flag which indicates whether we're in the **vom*** or the **voml*** ''half'' of the definition.

```
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)
          (pair (vom t m (car g)
                     (get (car g) v) (delete (car g) l) vec)
                (vom f m (cdr g) v l vec)
                (delete (car g) l))
          (init nil (length vec))))
```

Figure 6: The Oral Messages Algorithm—The Real Version

We suspect (and hope) that this discussion has increased your intuition about the functioning of the algorithm, at least for the case of four processors and one fault. An adequate intuition is quite difficult to develop, however, as the numbers of processors, faults, and rounds of message exchange increase. For any assurance that the algorithm achieves its goals, we need to state and prove its correctness formally.

3.2 The Correctness Theorems for the Algorithm

Recall from section 2.2 that an algorithm achieves interactive consistency if conditions IC1 and IC2 below are satisfied.

IC1. All loyal lieutenants obey the same order.

IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

In this section we display the theorems which establish that the formal analogues of IC1 and IC2 hold for the version of the Oral Messages algorithm described in the previous section.

The version of this correctness theorem given by Lamport, Shostak, and Pease [7] is:

THEOREM 1: *For any m, Algorithm OM(m) satisfies conditions IC1 and IC2 if there are more* *than 3m generals and at most m traitors.*

One issue is how to introduce the notion of a *faulty* process. We do this by declaring a predicate **`faulty`**, which takes as its argument a processor name. This assumes that a processor is either always faulty or always non-faulty; it does not distinguish transient or intermittent faults from permanent faults. We could do so but it would require some additional mechanism. Our choice of formalism does not imply that a faulty processor must somehow behave in a fashion which *other processes* can recognize as faulty. If it did, our algorithm could be much simpler—merely ignore any messages originating from or routed by faulty processors. The function **`fault-count`** applied to a list of processor names returns the number which are faulty; **`good-count`** is **`(difference (length l) (fault-count l))`**.

The theorem asserting that our **`vom`** function satisfies IC1 is shown in Figure 7. Recall that IC1 requires that all loyal lieutenants obey the same order. The function **`vom`** returns a vector in which each lieutenant's name is bound to the value it thinks is the general's value. We wish to formalize the notion that for any two loyal lieutenants, these values are the same. This is quite straightforward in our formalism, though we need a number of hypotheses to guarantee that the arguments are properly related.

```
Theorem. IC1
   (implies (and (setp l)                                   ; 1
                 (member i l)                               ; 2
                 (member j l)                               ; 3
                 (not (faulty i))                           ; 4
                 (not (faulty j))                           ; 5
                 (not (member g l))                         ; 6
                 (leq (times 3 m) (length l))               ; 7
                 (leq (fault-count (cons g l)) m)           ; 8
                 (bounded-number-listp l (length vec)))     ; 9
             (equal (get i (vom t m g v l vec))
                    (get j (vom t m g v l vec))))
```

Figure 7: Our Version of IC1

Executing our algorithm for **`m`** rounds of message exchange where general **`g`** has value **`v`** and **`l`** is the list of lieutenants is a call to **`(vom t m g v l vec)`**. The conclusion of lemma **`IC1`** asserts that the values computed for lieutenants **`i`** and **`j`** are equal. The hypotheses specify the conditions under which this can be shown to hold. Using the line numbers in Figure 7 for reference, these hypotheses assert that:

1. the list of lieutenants has no duplications;
2. lieutenant **`i`** is on this list;
3. lieutenant **`j`** is on this list;
4. lieutenant **`i`** is not faulty;
5. lieutenant **`j`** is not faulty;
6. the general **`g`** is not on the list of lieutenants;
7. $\mathbf{3} \times \mathbf{m}$ is less than or equal to the number of lieutenants;
8. there are at most **`m`** faulty processors;
9. all of the lieutenant names on **`l`** are legal indices into the vector **`vec`**.

We believe that this is a reasonable transcription of IC1 into our formalism.

Some of our hypotheses may seem unnecessary or arbitrary. For example, hypothesis 6 asserts that the general is not listed among the lieutenants. This is ‘‘obvious’’ from the description of the problem and is not stated explicitly in any of the journal proofs of IC1. However, it is necessary for the theorem to be valid. Such elucidation of implicit assumptions is one of the side benefits of fully formalizing such a theorem for mechanically aided proof.

We now consider IC2. Recall that IC2 asserts that if the commanding general is loyal, then every loyal lieutenant obeys the order he sends. In our formalization, this means that the value bound to the name of each non-faulty process must be exactly the value of the general. This is stated formally in the conclusion of lemma **`IC2`** in Figure 8.

```
Theorem. IC2
  (implies (and (setp l)                                ; 1
                (member i l)                            ; 2
                (not (faulty i))                        ; 3
                (not (member g l))                      ; 4
                (not (faulty g))                        ; 5
                (leq (times 3 m) (length l))            ; 6
                (leq (fault-count l) m)                 ; 7
                (bounded-number-listp l (length vec)))  ; 8
           (equal (get i (vom t m g v l vec))
                  v))
```

Figure 8: Our Version of IC2

The hypotheses of this lemma ensure that:

1. the list of lieutenants contains no duplicates;

2. lieutenant **`i`** is on the list;

3. **`i`** is not faulty;

4. the general **`g`** is not among the lieutenants;

5. **`g`** is not faulty;

6. **`3`** $\times$ **`m`** is less than or equal to the number of lieutenants;

7. the total number of faulty processors is less than or equal to **`m`**;

8. each lieutenant name is a legal index into **`vec`**.

3.3 Comments on the Proof of the Interactive Consistency Conditions

We will not review the proofs of the interactive consistency conditions here; except to make a few comments relating to the fact that the proofs have been machine checked. The complete script of Boyer-Moore ‘‘events’’ necessary to replay the proof is available upon request.

The proofs of lemmas **`IC1`** and **`IC2`** are a fairly difficult exercise in mechanical theorem proving. In one sense, there was no proof *discovery*; Lamport, Shostak, and Pease provide ‘‘journal level’’ proofs that their version of the algorithm satisfies IC1 and IC2. However, the gap between what is currently acceptable to even the best of mechanical theorem provers and to the mathematically sophisticated reader of a technical journal is still substantial. This does not imply, however, that the completely formal treatment required to render the proof acceptable to a mechanical proof checker is useless. We believe that we gained considerable insight into the Oral Messages algorithm from our formalization.

To illustrate this, consider the proof of the following lemma from [7]. This is the key lemma from which IC2 follows.

> LEMMA 1: *For any m and k, Algorithm OM(m) satisfies IC2 if there are more than $2k + m$ generals and at most k traitors.*
>
> PROOF. The proof is by induction on m. IC2 only specifies what must happen if the commander is loyal. It is easy to see that the trivial algorithm OM(0) works if the commander 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 commander sends a value v to all n-1 lieutenants. In step (2)[4], each loyal lieutenant applies OM(m-1) with n-1 generals. Since by hypothesis, $n > 2k+m$, we have $n\text{-}1 > 2k+(m\text{-}1)$, 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 $n\text{-}1 > 2k+(m\text{-}1) \geq k$, 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 $majority(v_1,...,v_{n\text{-}1}) = v$ in step (3), proving

[4]These steps refer to their description of the algorithm reproduced in Figure 1.

IC2.

Though seemingly straightforward, there is a considerable amount 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 those 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 IC2 was the primary effort in the proof. The final invariant is illustrated in Figure 9. We will not bother to describe some of the subsidiary concepts such as **`non-faulty-value`**. which are involved in the statement of the invariant. Suffice it to say that this theorem captures that key property maintained by **`vom`** which guarantees that it satisfies IC2. The corresponding invariant for IC1 is substantially more involved.

Advocates of the view that fully formal and machine checked proofs do not contribute materially to mathematics may feel that our formalization elucidates only detail which is better suppressed. We feel, however, that we understand the algorithm and the reason it works better for the effort. Moreover, we feel that a mechanically checked proof such as ours can eliminate errors which the much touted ‘‘social process’’ might overlook. This is particularly true for domains such as this where a well-developed intuition is difficult to cultivate.

3.4 Extending the Specification

Our function **`vom`** describes an ‘‘abstract implementation’’ of the *OM* algorithm which has been proved to meet its specification, i.e., to achieve interactive consistency in the presence of a limited number of faults. Our formulation of **`vom`** is, as much as possible, a direct transcription of the Pease, Shostak, and Lamport algorithm into the formalism of the Boyer-Moore logic.[5] Our ultimate goal, however, was a hardware implementation of this algorithm at a much lower level of abstraction and as part of a fault-tolerant

[5] An earlier version used mappings rather than vectors as the basic data structure. This version was perhaps slightly more abstract and closer to the published algorithm. However, we found it less amenable for mapping to a hardware implementation.

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 9: The Invariant for IC2

device.

Aimed in that direction, we need three intermediate steps at the specification level.

1. The formulation of **vom** given above takes the perspective of a group of processes trying to determine the private value of a single general. We require a formulation in which each process tries to determine the values of *all* of the other processes. This requires running n applications of *OM*.
2. As previously formulated, **vom** is a recursive function in which the parameter m determines the depth of recursion. This does not map well onto a lower level implementation in which the algorithm is implemented as a sequence of ‘‘steps.’’ To ease this mapping, we require a function which computes *traces* of the execution of the algorithm.
3. The abstract implementation was parameterized for an arbitrary number of processes. We wished to implement the instance in which $n = 4$ and $m = 1$.

3.4-A Multiple Applications of OM

To reach agreement, each process among a set of processes must act in turn as the general in an application of **`vom`**. As a step in that direction we define a subsidiary function **`om`** to simplify the argument list to **`vom`**, relying on the assumption that processes have indices in the range **`[0..n-1]`**. We then define the function **`oml`** recursively to apply **`om`** to each member of a list of process names in turn. Finally, we simplify the argument list of **`oml`** with the function **`omli`**.

Definition.
```
(om n g v m)
   =
(vom t m g v (delete g (indexlist n)) (init nil n))
```

Definition.
```
(oml l vec m)
   =
(if (listp l)
    (cons (put (car l)
               (get (car l) vec)
               (om (length vec) (car l)
                   (get (car l) vec) m))
          (oml (cdr l) vec m))
    nil)
```

Definition.
```
(omli vec m)
   =
(oml (indexlist (length vec)) vec m)
```

The function **`omli`** produces an **`n`** $\times$ **`n`** matrix in which the **`i`**th row contains everyone's guess at **`i`**'s local value. The **`i`**th column of the matrix is the *interactive consistency vector* for process *i*.[6] This vector contains the values which process **`i`** concludes are the local values for each of the other processes. In general, **`(get i (get g (omli vec m)))`** is the value which process **`i`** concludes is process **`g`**'s local value.

We can prove the following two facts about **`oml`** using the interactive consistency conditions proved of **`vom`**.

1. In the matrix value returned by **`oml`**, any two non-faulty processes agree on the local value of all other processes.
2. Each non-faulty process has the correct value for a non-faulty general.

These facts correspond to IC1 and IC2, respectively. The formal versions are displayed below.

[6] Since this matrix is really a list of lists, it is easier to fetch a ''row'' than a ''column.'' To extract the interactive consistency vector conveniently requires doing the equivalent of a matrix inversion.

```
Theorem. OMLI-IC1
  (implies (and (lessp g (length vec))
                (lessp i (length vec))
                (lessp j (length vec))
                (not (equal (fix i) (fix g)))
                (not (equal (fix j) (fix g)))
                (not (faulty (fix i)))
                (not (faulty (fix j)))
                (lessp (times 3 m) (length vec))
                (leq (fault-count (indexlist (length vec))) m))
           (equal (get i (get g (omli vec m)))
                  (get j (get g (omli vec m))))))
```

```
Theorem. OMLI-IC2
  (implies (and (lessp g (length vec))
                (lessp i (length vec))
                (not (faulty (fix g)))
                (not (faulty (fix i)))
                (lessp (times 3 m) (length vec))
                (leq (fault-count (indexlist (length vec))) m))
           (equal (get i (get g (omli vec m)))
                  (get g vec)))
```

From these two properties, we can prove that two non-faulty processes have identical interactive consistency vectors. Recalling that our ultimate goal is a fault-tolerant system of processors, we can be assured that the non-faulty processors generate identical actuator values if these are computing by applying the same filtering function to these interactive consistency vectors.

3.4-B Traces of *OM* Applications

The function **`omli`** formally describes a single instance of **`n`** processes reaching agreement through **`m`** rounds of information interchange. This formalization is not conducive to mapping down to a lower-level implementation which executes the algorithm in a number of ‘‘steps’’ and maintains a process *state*. Therefore, we define a *trace function* **`output*`** to model a collection of processes attempting to reach agreement through time. The input to **`output*`** is sequence of **`n`**-tuples of sensed values where each element of this sequence is a vector of sensor values. The function produces a sequence of vectors of actuator values.

At each step, the trace function applies a *step function* **`output`**. The input to this function is one of the input **`n`**-tuples and its result is one of the output vectors. The function **`output`** in turn involves an application of **`oml`**, and of a *filter function* which computes an output value based on an interactive consistency vector. An example of such a filter function is **`majority`**. The function **`output`** is defined as follows:

Definition.

```
(output vec m)
    =
(filterlist (matrix-invert (omli vec m)))
```

.

Here **`filterlist`** applies the **`filter`** function to each element in a list; **`matrix-invert`** inverts the data structure to put it into a format in which the interactive consistency vectors are readily accessible as described in footnote [6]. The trace function can now be written as follows:

Definition.

```
(output* senses m)
    =
(if (listp senses)
    (cons (output (car senses) m)
          (output* (cdr senses) m))
    nil)
```

The Byzantine properties of **`oml`** are provably inherited by the trace version. In particular, we can prove that, given a sufficiently small number of faulty processes, two non-faulty processes always agree on their outputs. Formally, this result is stated in the following lemma:

Theorem. **`OUTPUT*-FAULT-TOLERANT`**

```
(implies (and (numberp i)
              (numberp j)
              (lessp i n)
              (lessp j n)
              (not (faulty i))
              (not (faulty j))
              (tablep n senses)
              (lessp k (length senses))
              (lessp (times 3 m) n)
              (leq (fault-count (indexlist n)) m))
         (equal (get i (get k (output* senses m)))
                (get j (get k (output* senses m)))))
```

Notice that the trace-oriented version shifts the focus of attention from the matrix maintained by the *OM* algorithm to the actual outputs of the algorithm, i.e., to the signals going to the external actuators.

3.4-C Instantiating the Design

Our final step at the specification level is to decide how large a system of communicating processes we wish to implement. By instantiating the trace function with $n = 4$ and $m = 1$, we obtain a specification for a system of four redundant processes that achieve Byzantine agreement, and which can tolerate up to one faulty process. The architecture of this system is illustrated in Figure 10. It would be trivial to implement our specification with a different number of processors. However, the implementation details would be quite different.

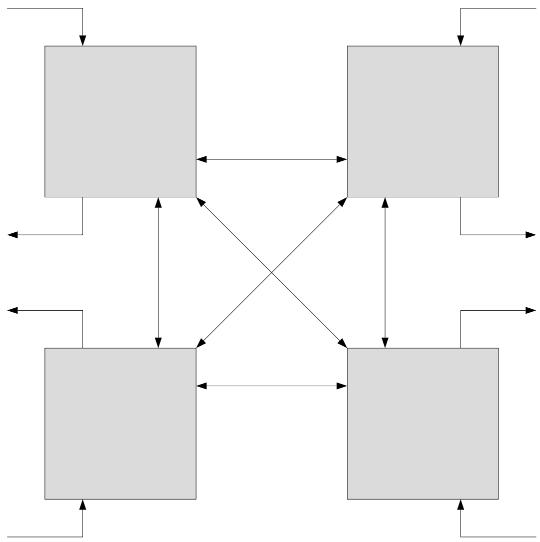

Figure 10: Four Redundant Processes

4. The Implementation and Its Proof

Implementing our circuit in hardware entails describing the internal logic of each of the four processes represented by the boxes in Figure 10 and explaining how these are connected to yield a fault-tolerant system. The processes achieve agreement after exchanging messages. To prove this fact, we show that our hardware implementation is a correct implementation of the abstract version described in the previous sections.

4.1 The Implementation

One goal of our design was for the four processes to be identical in order to minimize the amount of proof effort and to reduce the expense of constructing the system. Thus, in describing the design of a single process, we are actually describing the design of *each* of four processes.

Each process has five inputs: a sensor value, clock, and data lines from each of the other three processes. Additionally, each process has four outputs: an actuator and data lines out to each of the other processes. These inputs and outputs are listed below. In our formal description of the circuit the widths of the data paths are not fixed. This leaves the implementor free to choose a data width.

- **sense**: a sensed value.
- **clock**: a clock waveform.
- **data_in**: inputs from the three other processes.
- **data_out**: outputs to the three other processes.
- **actuator**: output to some actuator.

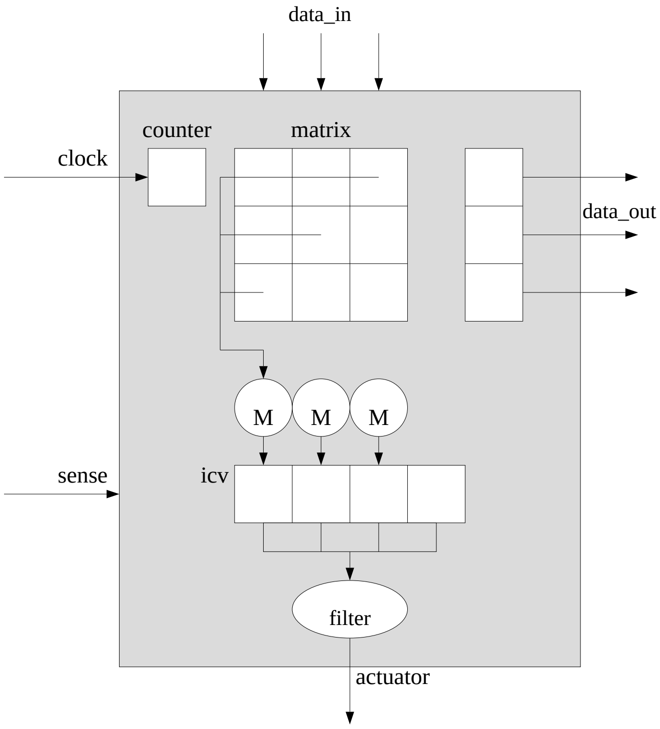

Figure 11: The Internal State of a Process

Figure 11 shows the *internal* state of a single process, along with some of the internal data paths. The internal state of a process contains the following components.

- **counter**: a 3-bit counter, used to cycle a process through 8 steps.
- **matrix**: a 3×3 matrix of data used to store values received during the information exchange.
- **icv**: the 1×4 interactive consistency vector for this process. **`ICV[3]`** holds the process's local value, derived from the sense input for that process.

The inter-connection of the processes to accomplish information exchange is depicted in Figure 12. Each arrow represents one-way communication. For each $i \in \{0, 1, 2, 3\}$, and $j \in \{0, 1, 2\}$, **`data_in`**[j] for process i is connected to **`data_out`**$[2 - j]$ of process $(i + j)$ *mod* 4. The interconnection scheme is designed to assure that all of the processes are identical.

One result of our desire for uniformity is that the interactive consistency vectors computed by two non-faulty processes are not actually identical, but are, in fact, rotations of one another. This implies that the filter function defined on the interactive consistency vector must be invariant under rotations of its vector argument.

Each process cycles through the 8 steps displayed in Figure 13. The purpose of each step is described below. The steps are numbered by the value of the 3-bit counter. The four processes share the clock input and hence perform these steps synchronously.

0. Read the sensed input. Save this as the process's local value in **`ICV[3]`**. Also, place this value on the output lines to the other three processes. This begins the report of each process's local value to all of the other processes.

1. Receive the local values of the other three processes, and store them in row **`0`** of the matrix.

2,3. Fill the remaining rows of each matrix with the reports of each process's value. In steps 2 and 3 each process receives two values from each of the other three processes. At the end of step 3, the information exchange required for the four instances of *OM*(1) is complete.

4. Compute the interactive consistency vector. This is accomplished by computing the majority of the three reported values for each of the other processes. (The circle labeled M in Figure 11 represents a 3-input majority circuit.)

5. Compute the actuator output based on the value of the interactive consistency vector. This is represented by a call to a function *filter*. In our specification *filter* is not defined, but is constrained to be invariant under rotation of its argument

6,7. No state change other than incrementing the counter.

This functionality is encoded in a function **`local-step`** which tests the clock value and updates the process state accordingly. The overall circuit is described by a function **`global-step`** which takes as input a list of four sensor values and a list of four process states and returns a list of the updated process states. Schematically, **`global-step`** is merely:

Definition.
`(global-step senses states)`
=
`(list` *<local step for process 0>*
<local step for process 1>
<local step for process 2>
*<local step for process 3>***`)`**

This is the step function for our system of four processors. The ‘‘interpreter’’ for this system is a function **`global-steps`** which repeats **`global-step`** repeatedly. A trace version **`c*`** is also defined which returns the list of actuator output 4-tuples produced after each global step. From the description above, it should be clear that the elements of this list change only every eight positions.

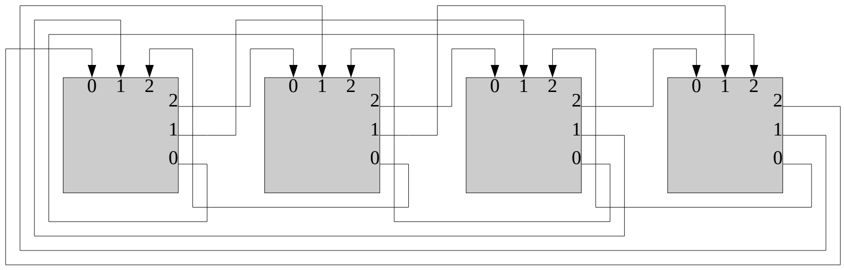

Figure 12: Process Interconnections

The behavior of our circuit can be summarized as follows. Each process senses its input and sends it out to each of the other processes, each process passes the values received on to the other processes, computes an interactive consistency vector on the basis of the values received, and then produces an actuator value by applying a filter function to this vector five steps after the input was sensed. Behavior is entirely synchronous among the four processors. The actuator value remains fixed until a new actuator value is computed on the next cycle. The process repeats as long as there are senses on the input lines.

4.2 The Proof of Correctness of the Implementation

The instance of the trace function **`output*`** described in Section 3 with $n = 4$ and $m = 1$ serves as a specification function for our circuit design. This function includes a call to *OM* to perform the information exchange. Because *OM* achieves interactive consistency, we have proved that at any point in the trace all non-faulty processes agree if there are a sufficient number of non-faulty processes.

Recall that **`output*`**, our trace function *at the specification level*, returns the actuator 4-tuples following a complete round of the *OM* algorithm. Similarly, our trace function **`c*`** *at the implementation level* projects out of the state of each process the value of its actuator after each global step. The appropriate relationship among **`global-step`**, **`c*`**, and **`output*`** is depicted in Figure 14.

Notice that the time granularity of **`c*`** is greater than that of **`output*`**. It takes **`c*`** five clock ticks to compute actuator values in response to a set of sense inputs. The intermediate steps of the trace are not of interest in the statement of interactive consistency. To relate the two traces, it is useful to define the notion

```
Case Counter:

0: data_out[i] ← sense, i ∈ {0, 1, 2}
   icv[3]      ← sense
   clock       ← clock+1

1: matrix[0,i] ← input[i], i ∈ {0, 1, 2}
   data_out[0] ← input[1]
   data_out[1] ← input[0]
   data_out[2] ← input[0]
   clock       ← clock+1

2: matrix[1,i] ← input[i], i ∈ {0, 1, 2}
   data_out[0] ← matrix[0,2]
   data_out[1] ← matrix[0,2]
   data_out[2] ← matrix[0,1]
   clock       ← clock+1

3: matrix[2,i] ← input[i], i ∈ {0, 1, 2}
   clock       ← clock+1

4: icv[0] ← majority(matrix[0,0], matrix[1,2], matrix[2,1])
   icv[1] ← majority(matrix[0,1], matrix[1,0], matrix[2,2])
   icv[2] ← majority(matrix[0,2], matrix[1,1], matrix[2,0])
   clock  ← clock+1

5: Actuator    ← filter(icv)
   clock       ← clock+1

6: clock       ← clock+1

7: clock       ← clock+1
```

Figure 13: Process Steps

of *n-selection*. The n-selection of **trace** is the list consisting of successive $(\mathbf{n}-1)^{st}$ elements of **trace**.

The proof of correctness of the circuit design requires the proof that some selection on **c*** equals the trace **output***. We have chosen $n = 7$ as the selector value in our proof. Figure 14 depicts the relationship proved between **c*** and **output***. We were able to formally establish the following equality relating the behavior of the circuit design to the specification function.

```
(equal (select 7 (c* senses states))
       (output* senses 1))
```

Recall the theorem **output*-fault-tolerant** (discussed in Section 3.4-B) which says that two non-faulty processes agree on their outputs. The conclusion of that theorem is that:

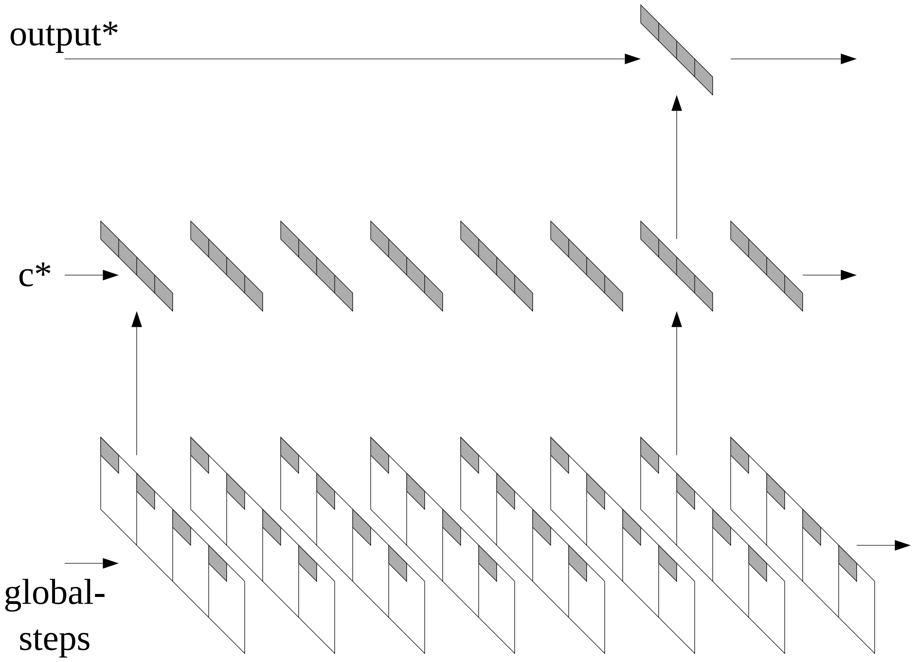

Figure 14: Correspondence among Trace Functions

```
(equal (get i (get k (output* senses m)))
       (get j (get k (output* senses m))))
```

Substituting **`(select 7 (c* senses states))`** into this lemma, with $n = 4$, $m = 1$ and $l = \{0, 1, 2, 3\}$ gives a theorem which says that the circuit design, as defined by **`c*`**, achieves agreement every 7th ‘‘tick’’ of the clock.

Theorem. **`IMPLEMENTATION-FAULT-TOLERANT`**

```
(implies
   (and (numberp i)
        (numberp j)
        (lessp i 4)
        (lessp j 4)
        (not (faulty i))
        (not (faulty j))
        (tablep 4 senses)
        (lessp k (length senses))
        (leq (fault-count (indexlist 4)) 1)
        (good-state-listp states)
        (equal (length states) 4)
        (equal (clocks states) (init 0 4)))
   (equal (get i (get k (select 7 (c* senses states))))
          (get j (get k (select 7 (c* senses states))))))
```

The last three hypotheses assure that the **`states`** parameter is properly formed and that the clocks in each local state are initially synchronized (set to 0). We take the proof of this theorem as a satisfactory formal demonstration of the correctness of the circuit design.

5. The Impossibility Result

The *OM* algorithm described in the previous sections achieves interactive consistency in the presence of m faults if there are at least $3m + 1$ total generals. An interesting fact proved by Pease, Shostak, and Lamport [8] is that this performance is *optimal* in the sense that no algorithm can achieve interactive consistency solely via message exchange if there are more faults.

We have formalized this result in the Boyer-Moore logic machine checked its proof. Proving this theorem mechanically in the Boyer-Moore theorem prover was an interesting challenge for several reasons.

- It has typically been quite difficult to prove a negative existential statement in the Boyer-Moore logic, except in cases where the range of the quantifier is inductively defined, which is not true here.
- Statement of the theorem requires consideration of some second order concepts. The Boyer-Moore logic is first order.

Our formal statement of the impossibility result used several features which have been added recently to the Boyer-Moore logic and illustrated a somewhat surprising versatility of the logic. We are aware of at least two previous proofs of impossibility results [2, 9] carried out in the logic. However, these results were specified and proved before the addition of these constructs which are currently available. Though we believe that this specification and proof could have been carried out without them, the use of the partial specification capability [4] and quantification and free variables [5] made the specification of the problem extremely natural.

5.1 Review of the Theorem

To state the problem, we find it convenient to quote rather extensively from [8].

> First, define a *scenario* as a mapping from the set P^+ of all nonempty strings over P, to V. For a given $p \in P$ define a *p-scenario* as a mapping from the subset of P^+, consisting of strings beginning with p, to V.

The appropriate intuition here is that processors attempt to achieve interactive consistency by exchanging their private values with other processes via messages. A value may be relayed through any number of processes. Intuitively, a scenario σ takes a string $p_i p_{i-1} ... p_2 p_1$ and returns the value that p_1 said that p_2 said that .... p_{i-1} said that p_i's private value was. For any nonfaulty processor q, we know that $\sigma(q)$ is q's private value.

A processor which is nonfaulty will faithfully relay any value received. This motivates the following definition.

> For a given choice $N \subseteq P$ of nonfaulty processors and a given scenario σ, say that σ *is consistent with* N if, for each $q \in N$, $p \in P$, and $w \in P^*$ (the set of all strings over P), $\sigma(pqw) = \sigma(qw)$. (In other words, σ is consistent with N if each processor in N always reports what it knows or hears truthfully.)

Now we are ready to define the notion of interactive consistency.

> For each $p \in P$, let F_p be a mapping which takes a p-scenario σ_p and a processor q as arguments and returns a value in V. (Intuitively, F_p gives the value that p computes for the element of the interactive consistency vector corresponding to q on the basis of σ_p.) We say that $\{F_p \mid p \in P\}$ assures *interactive consistency for m faults* if for each choice of $N \subseteq P$, $|N| \geq n - m$, and for each scenario σ consistent with N,
>
> (i) for all $p, q \in N$, $F_p(\sigma_p, q) = \sigma(q)$,
>
> (ii) for all $p, q \in N$, $r \in P$, $F_p(\sigma_p, r) = F_q(\sigma_q, r)$,
>
> where σ_p and σ_q denote the restrictions of σ to strings beginning with p and q, respectively.

It is helpful here to think of F as encoding some scheme by which the processors attempt to achieve a common view of each other's private values (the ‘‘interactive consistency vector’’). F_p is some computation that processor p performs based on the (arbitrarily large) collection of messages which arrive at p after being passed on by other processors.

Now this scheme (whatever it is) achieves interactive consistency if each nonfaulty processor learns the private value of each of the others and any two nonfaulty processors agree upon a value for each other processor in the system. If this third processor is faulty, we have no assurance that this common value is actually the private value of that process; it may be some ‘‘default’’ value.

The theorem from [8] which states our desired impossibility result is given below:

> THEOREM. *If $|V| \geq 2$ and $n \leq 3m$, there exists no $\{F_p \mid p \in P\}$ that assures interactive consistency for m faults.*[7]

---

[7] The theorem as stated here is actually false; interactive consistency is guaranteed if $|P| = n \leq 2$. Therefore, the theorem requires the additional hypothesis that $n > 3$. That Pease, Shostak, and Lamport intended this is evident from their proof.

5.2 An Informal Proof Sketch

The proof is a *reductio ad absurdum*. Consider a system of three communicating processors of which one is faulty. Recall that our goal is to define a general scheme that achieves interactive consistency among nonfaulty processors. This scheme must be such that in each case, the two nonfaulty processors must come to an agreement about the values of all three processors, where the values for the nonfaulty processors must be the *actual* private values for those processors. In other words, each nonfaulty process is computing a vector of three values, $[v_A, v_B, v_C]$ with the constraints that:

IC1. the vectors of the nonfaulty processors are identical;

IC2. if processor *X* is nonfaulty, the value recorded by each nonfaulty processor for *X* must be the actual private value of *X*.

Recall also that any nonfaulty processor may have no way of determining which of the other processors may be faulty.

Now consider the three diagrams labeled α, β, and σ in Figure 15. Each diagram shows three communicating processors *A*, *B*, and *C*. In each picture the shaded processor is only faulty processor among the three, and the labels on the arrows are *C*'s putative private value as relayed by the processor at the blunt end of the arrow.

Assume that we have an algorithm which can always achieve interactive consistency among three processors even if one of them is faulty. Consider scenario σ. By IC1, our scheme will achieve its goal only if processes *A* and *B* record the same value for processor *C*. However, from *A*'s perspective scenario σ is totally indistinguishable from scenario α. In α, *A* has no recourse but to record *v* as *C*'s value if constraint IC2 is to be satisfied. Therefore, given exactly the same information, *A* must also record *v* as *C*'s value in σ. Similarly, from *B*'s perspective σ is indistinguishable from scenario β. Consequently, *B* has no recourse but to record *v'* as *C*'s value in scenarios σ and β. Therefore, *A* and *B* record different values for *C* in scenario σ, in violation of IC1. Our assumption that we could devise an algorithm which would assure interactive consistency has led to a contradiction.

The informal proof sketch above is actually quite close to the idea of the formal proof. The basic notion is that if at least a third of the processors are faulty, we partition the processors into three sets of processors which collectively behave as *A*, *B*, and *C* above. We then define three scenario's which behave as α, β, and σ. The reader is invited to scrutinize the proof in [8], as our machine checked version follows very closely the proof given there.

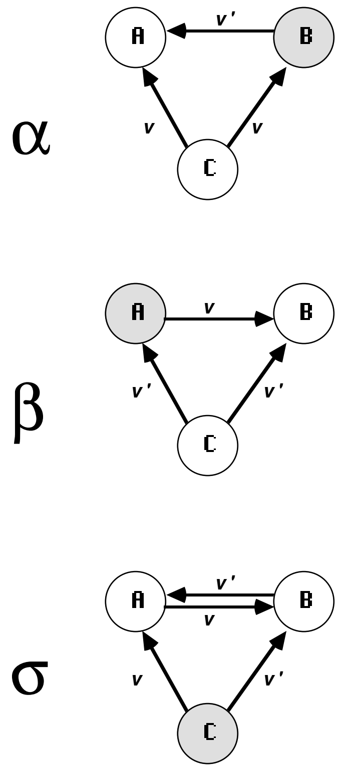

Figure 15: Three Scenarios

5.3 Specifying the Problem in the Boyer-Moore Logic

In this section we explain the specification of this problem in the Boyer-Moore logic [3]. The version of the logic used has been extended with facilities for handling quantification [5], free variables [6], and functional variables [4]. We assume the reader is familiar with this version of the logic.

Key notions in the statement of the formal proof are those of a *scenario* and of a *p-scenario*. We do not find it necessary to define these notions directly. Rather, we introduce the notion of the *application* of a scenario to a string to return a value. This is done with the Boyer-Moore **`constrain`** event displayed below.

Constrained Function Definition .

Introducing Functions: **apply-scenario, valuep, p-restrict.**

Constraints:
```
(and (valuep (apply-scenario sigma w))
   (implies (equal (car w) p)
            (equal (apply-scenario (p-restrict sigma p) w)
                   (apply-scenario sigma w)))
   (implies (and (valuep v1) (valuep v2))
            (equal
              (apply-scenario (list flg aa bb v1 v2) w)
              (abc flg aa bb v1 v2 w))))
```

Witness Functions:
```
apply-scenario: (lambda (x y) (abc (car x) (cadr x)
                                   (caddr x) (cadddr x)
                                   (caddddr x) y)))
valuep:         (lambda (x) t))
p-restrict:     (lambda (x y) x))))
```

This event introduces the new function symbols **apply-scenario**, **valuep**, and **p-restrict** along with certain axioms which govern them. Intuitively, **apply-scenario** takes to a scenario **sigma** and a string **w** and returns a value recognized by the function **valuep**. **p-restrict** takes a scenario **sigma** and a processor **p** and returns the p-scenario $\mathbf{sigma_p}$. Any application of $\mathbf{sigma_p}$ to a string beginning with **p** is identical to an application of **sigma** to that string.

The final axiom about **apply-scenario** refers to a specific defined function **abc** and is needed for the purposes of the proof. We will discuss it at length later, but basically it says that we are going to represent some specific scenarios by lists of a particular form and the application of these specific scenarios is captured by a function **abc** to be defined later.

The **constrain** event also supplies previously defined *witness* functions for the newly introduced function symbols which satisfy the axioms. This assures that the axioms are satisfiable and do not introduce an inconsistency into the theory. A constrain is only accepted by the theorem prover if it can be established that the proposed axioms are satisfied by the supplied witness functions.

Next we define the notion of a scenario σ being *consistent with N*, where *N* is a set of nonfaulty processors. This is defined via the *Skolemized Definition* event below. Such an event defines a concept involving explicit quantification by adding axioms representing the Skolemization of the event. (See [5] for details.) Notice that the definition **n-consistent** is very close to the definition of this concept in [8].

Skolemized Definition.

```
(n-consistent sigma N)
   =
(forall (w p q)
   (implies (member q N)
            (equal (apply-scenario sigma (cons p (cons q w)))
                   (apply-scenario sigma (cons q w)))))
```

To define our key concepts, we'll need the auxiliary notion of **`p-scenario-equivalence`**.

Skolemized Definition.

```
(p-scenario-equivalence sigma1 sigma2 p)
   =
(forall w
        (equal (apply-scenario sigma1 (cons p w))
               (apply-scenario sigma2 (cons p w))))
```

Two scenarios are p-equivalent iff they behave similarly on all strings beginning with ''*p*.'' This notion will be needed in the proof.

We are finally ready to define our main concept of an algorithm assuring interactive consistency for m faults. Since we would like to prove that under certain conditions no such algorithm exists, we need a very general characterization of the possible candidates for such an algorithm. We constrain a family of functions **`Ff`**. This family represents the collection of functions characterized in the notation of [8] as $\{F_p \mid p \in P\}$. **`Ff`** is conceptually a second order function which takes a **`p`**-scenario and a processor **`q`** and returns the private value which **`p`** computes for processor **`q`** on the basis of the information in the **`p`**-scenario. Thus, the call **`(Ff p sigma q)`** is our representation of the expression $F_p(\sigma_p, q)$ of the published proof.

The family **`Ff`** of functions is completely arbitrary, except that whenever two scenarios are **`p`**-equivalent they are indistinguishable from **`p`**'s perspective. It is this constraint which relates the notions of applying a scenario and the application of functions from **`Ff`**. Without it there is no formal tie between **`Ff`** and our intuition that **`Ff`** works by applying scenarios to strings to compute the private values of processes. **`Ff`** is introduced with the following **`constrain`** event.

Constrained Function Definition.

Introducing Function: `FF`.

Constraint:

```
(implies (p-scenario-equivalence sigma1 sigma2 p)
         (equal (FF p sigma1 q)
                (FF p sigma2 q)))
```

Witness Function:

```
FF: (lambda (x y z) f)
```

We are now ready to define the notion of a scheme which *assures interactive consistency for m faults*. This is introduced with the following event.

Skolemized Definition.

```
(Ff-consistent-for-m-faults Pp m)
   =
(forall (N sigma p q r)
  (implies (and (subset N Pp)
                (leq (length Pp) (plus (length N) m))
                (N-consistent sigma N)
                (member p N)
                (member q N)
                (member r Pp))
           (and (equal (Ff p (p-restrict sigma p) q)
                       (apply-scenario sigma (list q)))
                (equal (Ff p (p-restrict sigma p) r)
                       (Ff q (p-restrict sigma q) r)))))
```

This event defines a *conjecture* that **`Ff`** assures interactive consistency for the **`m`** faults in the set of processes **`Pp`**. We will prove that, under certain hypotheses about **`m`** and **`Pp`**, this conjecture is false.

Let's examine the definition **`Ff-consistent-for-m-faults`** more closely. The hypotheses of our conjecture are that we have a set **`N`** $\subseteq$ **`Pp`** of nonfaulty processors, that |**`Pp`**| $\leq$ |**`N`**| + **`m`**, and that we have a scenario **`sigma`** which is consistent with **`N`**. We also assume **`p`** $\in$ **`N`**, **`q`** $\in$ **`N`**, and **`r`** $\in$ **`Pp`**. Under these assumption, **`Ff`** assures interactive consistency for **`m`** faults if the conclusion is satisfied. Notice that the two conjuncts of the conclusion are exactly (i) and (ii) of the definition of interactive consistency for **`m`** faults of [8]. Namely,

$$F_p(\sigma_{\mathrm{p}}, q) = \sigma(q),$$

and

$$F_p(\sigma_{\mathrm{p}}, r) = F_q(\sigma_{\mathrm{q}}, r).$$

Given this definition/conjecture, it is straightforward to state the Boyer-Moore analogue of the impossibility result.

```
Theorem. IMPOSSIBILITY
  (implies (and (setp Pp)
                (partition Pp AA BB CC m)
                (leq m (length Pp))
                (leq (length Pp) (times 3 m))
                (valuep v1)
                (valuep v2)
                (not (equal v1 v2)))
           (not (Ff-consistent-for-m-faults Pp m)))
```

It asserts that under certain conditions on **Pp** and **m**, the assumption that **Ff** assures interactive consistency for **m** faults leads to a contradiction. These assumptions are the following:

1. **Pp** is a set (contains no duplicate process names);
2. **Pp** can be ‘‘partitioned’’ into three non-empty sets **AA**, **BB**, and **CC**, each of which have size less than or equal to **m**;
3. **m** ≤ |**Pp**|;
4. |**Pp**| ≤ **3m**;
5. there are at least two distinct values **v1** and **v2**.

Notice that the assumption that **Pp** can be partitioned into three nonempty subsets is implicit in the proof given in [8] but is certainly not implied by their statement of the theorem. The theorem as stated in [8] is false if |**Pp**| < 3, since interactive consistency can always be achieved in that case.

5.4 The Machine Checked Proof

The basic structure of the proof is very similar to that of the proof in [8] which is described as follows:

> Since $n \leq 3m$, *P* can be partitioned into three nonempty sets *A*, *B*, and *C* [it is here that we need our additional hypothesis that $n > 3$], each of which has no more than *m* members. Let *v* and *v'* be two distinct values in *V*. Our general plan is to construct three scenarios α, β, and σ such that α is consistent with $N = A \cup C$, β with $N = B \cup C$, and σ with $N = A \cup B$. The members of *C* will all be given private value *v* in α and *v'* in β. Moreover α, β, and σ will be constructed in such a way that no processor $a \in A$ can distinguish α from σ (i.e., $\alpha_a = \sigma_a$) and no processor *b* $\in$ *B* can distinguish β from σ (i.e., $\beta_b = \sigma_b$). It will then follow that for the scenario σ processors in *A* and *B* will compute different values for the members of *C*.

The three scenarios α, β, and σ are really three mutually recursive functions on $\mathbf{P^+}$. We define all three in the logic with the function **abc**, where the value of **flg** determines which of the three scenarios is being applied.

Definition.

```
(abc flg aa bb v1 v2 w)
   =
(if (and (listp w)
         (not (member (lastcar w) aa))
         (not (member (lastcar w) bb)))
    (let ((w1 (car w))
          (w2 (cadr w)))
      (case flg
            (a (if (listp (cdr w))
                   (if (listp (cddr w))
                       (if (member w2 bb)
                           (abc 'b aa bb v1 v2 (cdr w))
                         (abc 'a aa bb v1 v2 (cdr w)))
                     v1)
                 v1))
            (b (if (listp (cdr w))
                   (if (listp (cddr w))
                       (if (member w2 aa)
                           (abc 'a aa bb v1 v2 (cdr w))
                         (abc 'b aa bb v1 v2 (cdr w)))
                     v2)
                 v2))
            (otherwise
             (if (listp (cdr w))
                 (if (listp (cddr w))
                     (if (and (not (member w2 aa))
                              (not (member w2 bb)))
                         (if (member w1 aa)
                             (abc 'a aa bb v1 v2 (cdr w))
                           (abc 'b aa bb v1 v2 (cdr w)))
                       (abc 'c aa bb v1 v2 (cdr w)))
                   (if (member w1 bb)
                       v2
                     v1))
               v1))))
  v1)
```

We prove that the scenarios so defined are consistent with specific choices of **N**. For example, the following theorem shows that α is consistent with **N** = **A** $\cup$ **C**.

Theorem. ALPHA-CONSISTENT2

```
(implies (and (valuep v1)
              (valuep v2)
              (disjoint aa cc)
              (disjoint (append aa cc) bb))
         (n-consistent (list 'a aa bb v1 v2)
                       (append aa cc)))
```

The proof of **IMPOSSIBILITY** follows by expanding the quantified expression **(Ff-consistent-for-m-faults Pp m)** three times, once for each of the three scenarios α, β, and σ, and showing that this leads to a contradiction, in a fashion very analogous to the published proof.

Our machine checked version of the impossibility proof was primarily an interesting exercise in formal specification and mechanical theorem proving. We were pleased that our formal proof was quite similar to the published proof. This similarity was due in large part to constructs recently added to the Boyer-Moore logic and the accompanying support in the theorem prover. The use of the partial specification capability (**`constrain`** events) allowed the addition of axioms characterizing the second order concepts of application of a scenario in a provably consistent fashion. The use of quantification (via Skolemized definitions) made the definition of some intermediate concepts quite natural. Earlier versions of the Boyer-Moore logic would have required a purely constructive statement of the theorem; this would have been difficult or impossible to supply in this case.

6. Conclusion

We have verified a low-level hardware implementation of the Oral Messages algorithm of Pease, Shostak, and Lamport using a high-level abstract implementation as its specification. Because this abstract implementation has been formally proven to achieve interactive consistency, we are assured that our low-level implementation is fault-tolerant as well.

We have also machine checked the proof of an interesting impossibility result relating to fault-tolerance. This part of the work was primarily an exercise in formal specification and mechanical proof checking.

The main achievement of the work described in this paper is the demonstration of a fault-tolerant device that can be formally specified, and whose implementation can be proved correct. We have shown how to formally relate an abstract algorithm like *OM* to a design which is implementable in hardware. The main limitation of our device specification is that it does not explicitly account for distributed processes. Processes are described as operating synchronously. This simplifies the problem dramatically. Addressing this limitation is a future goal of our work.

All of the proofs, from the proof of correctness of the general Oral Messages Algorithm to the proof of the hardware implementation were fully machine checked. Proponents of the view that such fully formal and machine checked proofs do not contribute materially to mathematics or engineering may feel that our effort was superfluous.

From a mathematical perspective, we believe that two important goals of proof are to increase one’s understanding and intuition about the content and significance of a theorem, and to provide a convincing argument that it is, in fact, valid. Our proof efforts led us to develop a very clean and unambiguous statement of the algorithm and its correctness properties. We believe that we understand this quite subtle algorithm and the reason it works much better for the effort. Moreover, our success in convincing a congenitally skeptical mechanical proof checker of the validity of this theorem practically guarantees that we have eliminated any errors which the much touted ‘‘social process’’ might overlook. Such confidence is particularly comforting in domains such as fault-tolerant and real-time computing where a well-developed intuition is difficult to cultivate; the theorem prover is not subject to being misled by the urgings of a misguided or ill-informed intuition.

From an engineering perspective, we feel that our approach has several benefits. By proving properties such as the interactive consistency conditions with respect to our high-level abstract implementation, we retain the clarity and abstractness of the published algorithm and benefit from the intuitions derived from the published proof. By then mapping down to a more concrete characterization, but one which provably retains the fault-tolerant characteristics of the abstract version, we are able to derive a hardware level characterization of the algorithm which is trivial to implement. We suspect that an attempt to implement the Oral Messages algorithm directly from the published abstract presentation would be extremely error-prone.

References

1. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

2. Robert S. Boyer and J Strother Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. Technical Report ICSCA-CMP-37, Institute for Computing Science and Computer Appplications, University of Texas at Austin, 1983.

3. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

4. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore. Functional Instantiation in First Order Logic. Tech. Rept. 44, Computational Logic, Inc., May, 1989. Published in proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg, West Germany.

5. Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Tech. Rept. 43, Computational Logic, Inc., May, 1989.

6. M. Kaufmann. Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover. Tech. Rept. 42, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1990.

7. Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". *ACM TOPLAS 4*, 3 (July 1982), 382-401.

8. Marshall Pease, Robert Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". *JACM 27*, 2 (April 1980), 228-234.

9. N. Shankar. Checking the proof of Godel’s incompleteness theorem. Institute for Computing Science, University of Texas at Austin, 1986.

Table of Contents

List of Figures