Comparing Verification Systems: Interactive Consistency in ACL2

William D. Young
Computational Logic, Inc.
1717 W. 6th Street, Suite 290
Austin, Texas 78703
young@cli.com

March, 1997

Abstract

Achieving *interactive consistency* among processors in the presence of faults is an important problem in fault tolerant computing, first cleanly formulated by Lamport, Pease and Shostak and solved in selected cases with their Oral Messages (OM) Algorithm. Several machine-supported verifications of this algorithm have been presented, including a particularly elegant formulation and proof by John Rushby using EHDM and PVS. Rushby proposes interactive consistency as a benchmark problem for specification and verification systems. We present a formalization of the OM algorithm in the ACL2 logic and compare our formalization and proof to his. We draw some conclusions concerning the range of desirable features for verification systems. In particular, while higher-order functions, strong typing, lambda abstraction and full quantification have some value they come with a cost; moreover, many uses of such feature can be easily translated into simpler logical constructs which facilitate more automated proof discovery. We offer a cautionary note about comparing systems with respect to a small set of problems in a limited domain.

An earlier version of this paper appears in *Proceedings of the Eleventh Annual*

Conference on Computer Assurance, 1996, pp. 35–45. This work was supported by Computational Logic, Inc. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc. or any of its sponsors.

1 Introduction

Achieving *interactive consistency* among processors in the presence of faults is an important problem in fault tolerant computing, first cleanly formulated by Lamport, Shostak, and Pease[14, 9] and solved in selected cases with their Oral Messages (OM) Algorithm. Several mechanically checked verifications of this algorithm have been presented, including one by Bevier and Young[2] using Nqthm[4] and a particularly elegant formulation by Rushby using EHDM[15, 19] and PVS[12, 13][1]. Rushby draws some comparisons between his formulation and the earlier Bevier/Young version and between the automated tools used in the proofs. In particular, he cites the strong typing and higher-order functions available in PVS as helpful in leading to his improved formulation. He proposes interactive consistency as a potential benchmark for specification and verification systems.

Rushby's formalization of the Oral Messages algorithm is indeed significantly more elegant than the earlier Bevier/Young version and makes use of several features of EHDM and PVS not available in Nqthm, including higher order functions, lambda expressions, and full quantification. His proof is also quite different, and closer to the journal proof. We cannot dispute Rushby's claim that he found various features helpful in his thinking about the problem. But one might infer also from his success in formally analyzing the Oral Messages algorithm and other fault tolerant algorithms [12] that the features of EHDM and PVS highlighted by these efforts are either necessary or highly desirable in a modern specification and proof system. It is that implied suggestion that we question.

We have duplicated Rushby's formalization and proof of the Oral Messages algorithm using ACL2[7, 11], the successor to Nqthm. The formalization of the algorithm in ACL2, the statement of its correctness properties and their proof are very similar to Rushby's. We make the argument that the advantages of Rushby's version over the earlier Bevier/Young proof are due much more to cleverness in the formalization and proof than to any strengths of the logic or automated proof system. The point is not to disparage the

[1] Rushby's original proof used EHDM; he subsequently redid the proof using PVS. Though the proof styles in the two provers are quite different, the formulations of the problem are essentially identical. Consequently, any remarks we make about the *specification* of the problem will apply to either system. We will direct remarks concerning the *proof* to the PVS version, since EHDM is obsolete.

insight and fine work of John Rushby and his colleagues in tackling difficult problems in fault tolerance, but to offer a cautionary note on comparing proof systems with respect to any small problem or set of problems within a particular domain.

In the following sections, we describe the problem solved by the Oral Messages algorithm, sketch Rushby's formalization of the algorithm, present our formalization in the ACL2 logic and compare our formalization and proof to Rushby's. We draw some conclusions concerning the range of desirable features for verification systems. A small warning is in order: it is not among the goals of this paper to teach the details of the Oral Messages algorithm, of ACL2, or of any other logic. That material is much better covered elsewhere. However, we feel that one can follow our discussion without having considerable knowledge in those areas.

2 Interactive Consistency and OM

The problem addressed by the Oral Messages 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[9] describe the problem in terms of the colorful metaphor of Byzantine Generals attempting to arrive at a common battle plan through an exchange of messages. One or more may be traitorous and attempt to thwart the loyal generals by preventing them from reaching agreement.[2]

Lamport, et al. phrase the problem in terms of a single commanding general communicating with a number of lieutenant generals. The goal is 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.

[2] The notion of a "Byzantine" fault, which arises from their work, means that a faulty process may be *actively* malicious, attempting to thwart agreement by sending inconsistent values to the other processors.

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.

The generals' 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.

Notice that we are assuming synchronous systems in all of our discussions.

The Oral Messages algorithm $OM(m)$ describes m rounds of communication among the generals and is inductively defined for all nonnegative integers m. The goal is to convey an order by the commanding general to each of $n-1$ lieutenants. The description of the algorithm from [9] is as follows:

Algorithm OM (0)

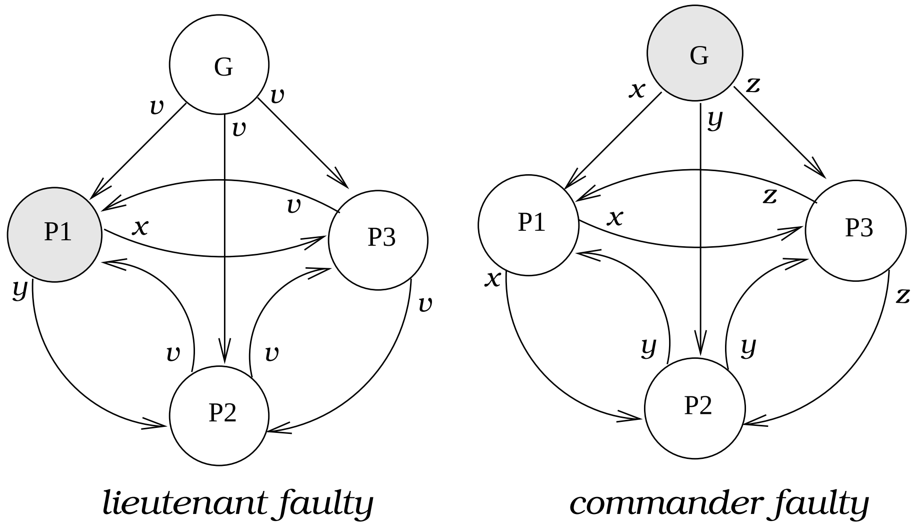
1. The commander sends his value to every lieutenant.
2. Each lieutenant uses the value he receives from the commander, or uses the value *retreat* if he receives no value.

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 $majority(v_1, \ldots, v_{n-1})$.

This version of the algorithm is "asymmetric" in the sense that the commanding general is in a distinguished position with respect to the lieutenants. There is also a symmetric version of the algorithm in which all processors are treated similarly. In the symmetric version, there is no distinguished commander. Rather, each processor has a value which to broadcast to each of the other $n-1$ generals. Each processor runs the same algorithm and computes a vector of values, one for each process. Rushby calls the symmetric version OMIC (Oral Messages, Interactive Consistency) and the asymmetric version OMBG (Oral Messages, Byzantine Generals).

To see how the algorithm works, consider the simplest interesting scenario: four processors, a commanding general and three lieutenants, where one of the four is traitorous. In this case, the algorithm requires one round of exchanges. The commander first sends an order to the three lieutenants. Then, each lieutenant relays the value received to each of the other lieutenants. There are two possibilities: the traitor is one of the three lieutenants, or is the commander.



Suppose the traitorous party is one of the three lieutenants, say P1. This means that the loyal commander has broadcast the same order to each of the three. The traitor may relay arbitrary values (or no value at all) to the other lieutenants. However, each of the two loyal lieutenants receives one reliable value from the commander and another relayed by the other loyal lieutenant. Taking the majority of the three values received, the loyal lieutenants arrive at a consistent shared view of the general's order, and one which corresponds to the order actually sent. Thus, IC1 and IC2 are satisfied. This scenario is illustrated in the left hand side of the diagram above.

Now, suppose the commanding general is the traitorous party, and all lieutenants are loyal. The lieutenants may receive arbitrary values from the commander, which they faithfully relay to their comrades. Each of the three lieutenants receives the same three values, one from the commander and one from each of the other two lieutenants. If there is a majority, they each compute the same value; if none, each records the default value of *retreat*. Again, IC1 and IC2 are satisfied. The right hand side of the diagram above illustrates this scenario.

These two scenarios illustrate how the loyal lieutenants can arrive at a consistent view of the general's value via this single exchange of messages provided that there is no more than one traitor among the four processors. In general, with n processors of which m are faulty, the OM algorithm achieves interactive consistency with m exchanges assuming $n \geq 3m + 1$. Proving this fact, by induction on m, is the goal of each of the formalizations we will consider.

Lamport, et al. also proved that the OM algorithm is optimal among its class of algorithms. That is, no algorithm relying solely on oral message passing for interprocess communication can achieve interactive consistency among a collection of processors if at least a third of the total are faulty.

3 Previous Formalizations of OM

Bevier and Young[2] published a formalization and proof of the Oral Messages algorithm using the Boyer-Moore Nqthm[4] theorem prover in the context of a larger project of building a verified circuit implementing interactive consistency [1, 10]. They also mechanically checked the Lamport, et al. proof that the Oral Messages algorithm is "optimal" among its class of algorithms. Bevier and Young reported of their proof:

> 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.

The Bevier/Young version used a rather awkward mutually recursive formalization on arrays and a complex invariant for the recursive cases. Moreover, their proof followed the purely constructive style common in the quantifier-free logic of Nqthm.[3] This led Rushby to observe that he "continued to find the journal description and proof more compelling than their formal version" [15] and to develop an alternative formulation and proof of the algorithm using EHDM.

Of course, in a very real sense a machine checked formalization does not *need* to be as compelling as a journal version, if the prover is sound. It is sufficient for the *problem formulation* to be clear and accurate, assuming we trust the machine-checked proof. Rushby himself argues strongly[16] that machine-checked proofs are required for fault-tolerant algorithms, such as OM, precisely because of their extreme subtlety and the tendency to get informal proofs wrong. However, the cleaner the formulation of the algorithm, the higher the likelihood that it will be scrutinized carefully and understood well by potential users and implementors. Therefore, even a machine certified proof will be most useful if it conveys insight.

Rushby reformulated the algorithm using first the EHDM system[19] and later PVS[12]. His formulation[15] is decidedly simpler and more elegant than the version by Bevier and Young. Rushby "found that one of the keys to simplifying the algorithm was to focus on the symmetric formulation (which is actually the form required), rather than the asymmetric Byzantine Generals form" [12] used by Bevier and Young. For completeness Rushby included formalizations of both the symmetric (OMIC) and asymmetric (OMBG) versions of the algorithm. He reports that his formalization of OMBG is due to his colleague Shankar, and that its formal verification "is very similar to that of OMIC, and was derived from that of OMIC in less than a day" [15]. However, he also observes 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," which is the route

[3]A limited version of quantification was available in Nqthm [3], but was not used in their proof of the algorithm correctness. Quantification was used in the proof of the optimality result.

```
distr = (λ r, t, p : (λ z : send (r, t, p, z))

OMIC (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
```

Figure 1: Rushby's Formalization of OMIC

followed by Bevier and Young. In the current paper, we will concentrate on the formalization of the OMIC version.

Rushby's version of the symmetric (OMIC) algorithm is given in Figure 1. Notice that it describes a higher order function which returns a function from processors to vectors of values. A vector, in turn, is a mapping from processors to values. Conceptually, OMIC $(r, v, caucus)(p)(q)$ gives p's opinion of q's value after r rounds of the algorithm, where v is the vector of initial values of all processes and *caucus* is the set of processors under consideration. The text in Figure 1 is neither raw EHDM nor PVS syntax. However, it (or something very like it) can be generated automatically from the PVS source using LaTeX.

Rushby's formulation of the interactive consistency conditions IC1 and IC2 for OMIC are given in Figure 2. These are not the final theorems, but rather the interesting inductive versions of the conditions. They are introduced first as universally quantified notions; lemmas are then introduced asserting that these conditions actually hold for arbitrary parameters. The final theorems on the full set of processes follow immediately from the inductive versions. It is often necessary, as in this case, to prove a stronger inductive theorem than the theorem actually desired. We do not include the OMBG version of the algorithm, which was the version used in the original Bevier/Young formalization.

In comparing his formulation to the Bevier/Young version, Rushby makes the following points:

- A significant improvement was gained by using the symmetric formulation of the problem, rather than the asymmetric version used by Bevier and Young.
- The richer specification capabilities available in EHDM and PVS avoided the need for the mutual recursion in the Nqthm script.
- The design decisions of EHDM (and presumably of PVS), such as strong typing, lambda abstraction and higher order logic, led to a much more perspicuous formalization.

According to Rushby, the upshot was that: "The complexity discovered by Bevier and Young was an artifact of their formalization and of the theorem prover at their disposal" [15].

$$
\begin{array}{l}
\mathrm{C1prop}\ (r)\ = \\
\quad (\forall\ p,\ q,\ y,\ caucus,\ v: \\
\qquad\qquad \mathrm{ok}\,(p)\ \wedge\ \mathrm{ok}\,(q)\ \wedge\ p\ \in\ caucus \\
\qquad\quad \wedge\ q\ \in\ caucus\ \wedge\ y\ \in\ caucus \\
\qquad\quad \wedge\ |caucus|\ >\ 3 * r \\
\qquad\quad \wedge\ r\ \geq\ |\mathrm{faulty\_members}\,(caucus)| \\
\qquad \supset\qquad \mathrm{OMIC}\,(r,\ v,\ caucus)(p)(y) \\
\qquad\qquad =\ \mathrm{OMIC}\,(r,\ v,\ caucus)(q)(y))
\end{array}
$$

C1: **Lemma** C1prop$\,(r)$

$$
\begin{array}{l}
\mathrm{C2prop}\ (r)\ = \\
\quad (\forall\ p,\ q,\ caucus,\ v: \\
\qquad\qquad \mathrm{ok}\,(p)\ \wedge\ \mathrm{ok}\,(q)\ \wedge\ p\ \in\ caucus \\
\qquad\quad \wedge\ q\ \in\ caucus \\
\qquad\quad \wedge\ |caucus|\ >\ 2 * |\mathrm{faulty\_members}\,(caucus)| + r \\
\qquad \supset\ \mathrm{OMIC}\,(r,\ v,\ caucus)(p)(q)\ =\ v(q))
\end{array}
$$

C2: **Lemma** C2prop$\,(r)$

Figure 2: Rushby's Version of IC1 and IC2 for OMIC

In the following, we would like to examine the question of how much of the improvement is really due to the different formalization and how much to the choice of logic and theorem prover. To do so, we first briefly describe ACL2, the successor to Nqthm, and then provide a translation of Rushby's OMIC formalization into ACL2.

4 Brief Description of ACL2

ACL2 is an extended, reimplemented version of Nqthm[4] that reasons about an applicative subset of Common Lisp. It has been designed and implemented by J Moore and Matt Kaufmann at CLI, with assistance and inspiration from Bob Boyer. The system is described more fully elsewhere in this issue[11].

By formalizing a logic around applicative Common Lisp the designers take advantage of the exceptionally good optimizing compilers for Common Lisp to get, in many cases, execution speeds comparable to C. Two guiding tenets of the ACL2 project have been to conform to all compliant Common Lisp implementations and to add nothing to the logic that violates the understanding that the user's input can be submitted directly to a Common Lisp compiler and then executed in an environment where suitable ACL2-specific macros and functions are defined.

4.1 The ACL2 Logic

The ACL2 logic is a first order logic of recursive functions providing mathematical induction on the ordinals up to ϵ_0 and two extension principles: one for recursive definition and one for constrained introduction of new function symbols. Each preserves the consistency of the extended logic. Examples of each are given in Section 4.3 below.

The syntax of ACL2 is that of Common Lisp. In addition, ACL2 supports the macro facility of Common Lisp. Macros provide a powerful abbreviation facility and can be used to make specifications much more succinct and easier to grasp. Because Common Lisp macros use functions to compute their results, macros provide an extremely powerful extension facility to the syntax of the language.

The following primitive data types are axiomatized: rational and complex numbers, character objects, strings, symbols, and lists. Common Lisp

`nil`	the empty list or "false" in Boolean contexts
`(if x y z)`	if `x` is not equal to `nil` return `y`, else `z`
`(equal x y)`	are `x` and `y` the same value?
`(car l)`	the first element of list `l`
`(cdr l)`	all but the first element of list `l`
`(cons x l)`	add `x` onto the front of list `l`
`(endp x l)`	is `l` empty or a non-list?
`(zp x)`	is `x` equal 0 or not a natural number?
`(remove x l)`	delete element `x` from list `l`
`(member-equal x l)`	is `x` a member of list `l`?

Figure 3: Some Elementary ACL2 Functions

functions on the above data types are axiomatized or defined as functions or macros in ACL2. Approximately 150 such symbols are axiomatized.

Several of the elementary ACL2 functions used in our specification are listed in Figure 3. Notice that, with the exception of `if`, all of these functions evaluate all of their arguments. Function `if` evaluates only its test and the selected branch. Some other functions, such as the elementary logical operators `not`, `and`, `or` and `implies`, are defined in terms of `if`.

4.2 Theorem Prover

The ACL2 theorem prover is a reimplementation of the Nqthm theorem prover[4] for the ACL2 logic. Every proof technique of Nqthm has been implemented in ACL2. Many have been extended significantly.

The prover has sophisticated heuristics controlling the application of the following proof techniques: preprocessing including tautology checking using ordered binary decision diagrams (OBDDs) if directed by the user; simplification by primitive type checking and rewriting; destructor elimination; cross-fertilization using equivalence reasoning; generalization; elimination of irrelevant hypotheses; and mathematical induction.

ACL2's data base is a hierarchical collection of definitions and theorems grouped into "books." A book can be included by reference into other books and multiple books can be read to construct the active data base. Books allow multiple users to share one another's work. Books can also define "theories" and theory manipulation functions. When two books are loaded together, checks are done to insure their logical compatibility.

ACL2 has numerous other features and facilities. Users of Nqthm will find the style of interaction with ACL2 very familiar, though the additional features make it a more powerful and versatile tool.

4.3 Some ACL2 Examples

The following is an example of a typical ACL2 function definition, in this case a recursive function which takes a list `l` of processors and returns a list of those which are "faulty." It behaves as follows: If we have reached the end of the list, return the empty list (`nil`). Otherwise, if the first element of the list satisfies the previously defined predicate `ok`, recur on the remainder of the list. If not, add that element onto the result of recurring on the remainder.

```
(defun faulty-members (l)
  (if (endp l)
      nil
    (if (ok (car l))
        (faulty-members (cdr l))
      (cons (car l) (faulty-members (cdr l))))))
```

Assuming that `ok` is a previously introduced executable function, this form both defines an executable Common Lisp function and a logical definition within the ACL2 logic.

A trivial property of this definition is expressed in the following ACL2 theorem:

```
(defthm faulty-member-not-ok
  (implies (member-equal p (faulty-members caucus))
           (not (ok p))))
```

It asserts that any element of the faulty-members of a list `caucus` does not satisfy the predicate `ok`. ACL2 proves this by induction, generating an appropriate induction schema from the definition of the function `faulty-members`. This theorem later may be invoked automatically as a conditional rewrite rule to reduce an occurrence of `(ok x)` to `nil`, assuming that the hypothesis can be relieved in context.

ACL2 also supports the introduction of axiomatically constrained function symbols which have no executable counterpart. For example, the following form introduces a new function symbol `send` of four arguments.

```
(encapsulate
  ((send (r v p q) t))

  (local (defun send (r v p q) v))

  (defthm send-reduction1
    (implies (and (ok p) (ok q))
             (equal (send r v p q) v))))
```

The function `send` is constrained to satisfy an axiom asserting that, if its third and fourth arguments are the names of `ok` processors, the value returned is its second argument. We are assured that adding this axiom maintains consistency because we are required to exhibit a "witness" function—the supplied *local* definition of `send`—which is used in proving the axiom. Only the name of the introduced function, its arity, and the axiom `send-reduction1` are visible outside the encapsulate form.

For readers put off by the Lisp prefix syntax, notice that these definitions could just as easily be presented in a non-Lisp syntax, perhaps such as the following:[4]

Definition 4.1

$$\text{faulty\_members}\,(l) \quad =$$

$$\begin{cases} \langle\rangle, & \text{if } \mathrm{l} = \langle\rangle; \\ \text{faulty\_members}\,(l) & \text{if } \mathrm{l} = x \circ l' \ \wedge \ \text{ok}\,(x); \\ x \circ \text{faulty\_members}\,(l) & \text{if } \mathrm{l} = x \circ l' \ \wedge \ \neg\text{ok}\,(x). \end{cases}$$

Such a notation probably has benefit when presenting the work to a non-Lisp-literate audience, and it can be generated automatically from Lisp code.[5] But it is potentially ambiguous, significantly harder to parse mechanically than the Lisp syntax, and may be a detriment when attempting to understand the

[4] We are using the notation $x \circ l$ to denote `(cons x l)`.

[5] ACL2 has an associated facility for generating LaTeX output in a slightly different format than what is shown here.

algorithm and devise a correct implementation for anyone who has expended the minimal effort to learn Lisp syntax. We'll return to the question of syntax in Section 6.

5 Formalizing OM in ACL2

Our goal in revisiting the Oral Messages algorithm with ACL2 was not to contribute a new formalization, but to evaluate how ACL2 performed given the elegant formalization discovered by Rushby. In this section we describe our formalization of the OMIC version of the algorithm. We have also carried out a formalization and proof of the OMBG version of the algorithm. However, there were no additional interesting lessons gained from that exercise, so we will concentrate on the OMIC version.

Studying Figure 1, we see that OMIC generates a finite mapping with signature

$$\text{processors} \rightarrow [\text{processors} \rightarrow \text{values}].$$

For example, for the case of $r = 0$, this mapping is:

$$p_0 \mapsto \left\{ \begin{array}{lcl} q_0 & \mapsto & \text{send}\,(0,\ v(q_0),\ q_0, p_0) \\ & \vdots & \\ q_k & \mapsto & \text{send}\,(0,\ v(q_k),\ q_k, p_0) \end{array} \right.$$

$$\vdots$$

$$p_k \mapsto \left\{ \begin{array}{lcl} q_0 & \mapsto & \text{send}\,(0,\ v(q_0),\ q_0, p_k) \\ & \vdots & \\ q_k & \mapsto & \text{send}\,(0,\ v(q_k),\ q_k, p_k) \end{array} \right.$$

where the p_i's and q_i's are drawn from *caucus*.

Notice, however, that the correctness conditions always refer to specific values of this higher order function. That is, instead of OMIC $(r,\ v,\ caucus)$, the interactive consistency conditions always make reference to a vector of values OMIC $(r,\ v,\ caucus)(p)$ or to a specific value OMIC $(r,\ v,\ caucus)(p)(q)$. Thus, rather than attempting to simulate Rushby's higher order function directly, we can "uncurry" his higher order expressions and define a pair of mutually recursive functions that compute these two expressions directly.

```
(mutual-recursion

 (defun omic (r v caucus p q)
   (if (and (member-equal p caucus)
            (member-equal q caucus))
       (if (zp r)
           (send 0 (binding q v) q p)
         (if (equal p q)
             (send r (binding q v) q q)
           (maj (remove q caucus)
                (omic1 (1- r)
                       (distr r (binding q v) q)
                       (remove q caucus)
                       p
                       (remove q caucus)))))
     (undef)))

 (defun omic1 (r v caucus p qlist)
   (if (endp qlist)
       nil
     (let ((q (car qlist)))
       (cons (cons q (omic r v caucus p q))
             (omic1 r v caucus p (cdr qlist)))))))
```

Figure 4: The ACL2 Version of OMIC

The result is a first order formalization of the OMIC algorithm that compares favorably in complexity to Rushby's version. The formalization in ACL2 notation is given in Figure 4. In the interest of factoring out the question of syntax, an equivalent version in a more conventional syntax is given in Figure 5.

Here we are defining two functions `OMIC` and `OMIC1` via a straightforward mutual recursion. A call to `OMIC (r, v, caucus, p, q)` computes the same value as Rushby's expression:

$$\text{OMIC}\,(r,\, v,\, \mathit{caucus})(p)(q).$$

A call to `OMIC1 (r, v, caucus, p, lst)` iterates `OMIC` over the values of

$$
\begin{array}{l}
\mathrm{OMIC}\,(r,\ v,\ \mathit{caucus},\ p,\ q)\ =\\
\quad \mathbf{if}\ p\ \in\ \mathit{caucus}\ \wedge\ q\ \in\ \mathit{caucus}\\
\quad\quad \mathbf{then\ if}\ r\ =\ 0\\
\quad\quad\quad \mathbf{then}\ \mathrm{send}\,(r,\ v(q),\ q,\ p)\\
\quad\quad\quad \mathbf{else\ if}\ p\ =\ q\\
\quad\quad\quad\quad \mathbf{then}\ \mathrm{send}\,(r,\ v(q),\ q,\ q)\\
\quad\quad\quad\quad \mathbf{else}\ \mathrm{maj}\,(\mathit{caucus}-\{q\},\\
\quad\quad\quad\quad\quad\quad \mathrm{OMIC1}\,(r-1,\ \mathrm{distr}\,(r, v(q), q),\ \mathit{caucus}\text{-}\{q\},\ p,\ \mathit{caucus}\text{-}\{q\}))\\
\quad\quad \mathbf{else}\ \mathrm{undef}\\
\quad \mathbf{end\ if}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{OMIC1}\,(r,\ v,\ \mathit{caucus},\ p,\ \mathit{lst})\ =\\
\quad \mathbf{if}\ \mathit{lst}\ =\ \langle\rangle\\
\quad\quad \mathbf{then}\ \langle\rangle\\
\quad\quad \mathbf{else}\ \ \mathbf{let}\ \mathit{lst}\ =\ q \circ \mathit{lst}'\ \ \mathbf{in}\\
\quad\quad\quad \langle q,\ \mathrm{OMIC}\,(r,\ v,\ \mathit{caucus},\ p,\ q)\rangle \circ \mathrm{OMIC1}\,(r,\ v,\ \mathit{caucus},\ p,\ \mathit{lst}')
\end{array}
$$

Figure 5: Our Formalization of OMIC in Conventional Notation

the list argument `lst`. For an appropriate choice of `lst`, this is effectively equivalent to

$$\lambda\, q :\ \mathrm{OMIC}\,(r,\ v,\ \mathit{caucus})(p)(q).$$

One explanatory comment is in order. We model finite mappings in Lisp using an *association list* or "alist," i.e., a list of pairs associating keys with values. We defined `(binding k v)` to return the value bound to key `k` in alist `v`, or `(undef)` if there is no such binding. It is not misleading to read `(binding k v)` as `v(k)`.

Even to the unpracticed eye, the ACL2 version no more complex than the PVS version and, for many readers, it may even be substantially simpler since it does not involve the higher-order constructs and lambda expressions in Rushby's version.[6] Certainly, for a practiced Lisp programmer familiar

[6] In truth, our first version was much more complicated because it simulated the computation of the higher order function. A reviewer suggested computing the required values directly, rather than computing the two dimensional array of values suggested by the higher order construction. The change was easy and it greatly simplifies the specification.

```
(defthm OMIC-PROP1
  (implies (zp r)
           (equal (omic r v caucus p q)
                  (if (and (member-equal q caucus)
                           (member-equal p caucus))
                      (send 0 (binding q v) q p)
                    (undef)))))

(defthm OMIC-PROP2
  (implies (not (zp r))
           (equal (omic r v caucus p q)
                  (if (and (member-equal p caucus)
                           (member-equal q caucus))
                      (if (equal p q)
                          (send r (binding q v) q q)
                        (maj (remove q caucus)
                             (omic1 (- r 1)
                                    (distr r (binding q v) q)
                                    (remove q caucus)
                                    p
                                    (remove q caucus))))
                    (undef)))))
```

Figure 6: Properties of OMIC

with standard recursive list programming idioms, the translation is straightforward. Our original coding of Rushby's algorithm took about two hours. It has gone through various simplifying revisions, each of which also took only a few hours. We believe any competent Lisp programmer could have carried out the translation, given the requisite insight regarding "uncurrying" the higher order function.

What confidence do we have that this formulation is actually equivalent to Rushby's version? Using the ACL2 prover, we have easily proved the lemmas in Figure 6. In more traditional mathematical notation, these can be rendered as follows:

Theorem 5.1 (OMIC Property1)

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

Theorem 5.2 (OMIC Property2)

$$
\begin{array}{l}
\quad r \neq 0 \\
\Rightarrow \\
\qquad \mathrm{OMIC}(r,\ v,\ \mathit{caucus},\ p,\ q) \\
\quad = \ \textbf{if } p \in \mathit{caucus} \wedge q \in \mathit{caucus} \\
\qquad\qquad \textbf{then} \\
\qquad\qquad\quad \textbf{if } p = q \\
\qquad\qquad\qquad \textbf{then } \mathrm{send}(r,\ v(q),\ q,\ q) \\
\qquad\qquad\qquad \textbf{else } \mathrm{maj}(\mathit{caucus} - \{q\}, \\
\qquad\qquad\qquad\qquad\qquad \mathrm{omic1}(r-1,\ \mathrm{distr}(r,\ v(q),\ q), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{caucus}{-}\{q\},\ p,\ \mathit{caucus}{-}\{q\}\ )) \\
\qquad\qquad\quad \textbf{end if} \\
\qquad\qquad \textbf{else } \texttt{undef} \\
\qquad\ \ \textbf{end if}
\end{array}
$$

Modulo the trick of "uncurrying" the higher order function calls, these theorems show that our version of the algorithm returns precisely the same value as Rushby's. In fact, they correspond almost exactly to Rushby's lemmas OM0_prop and OM_prop, which he (and we) used in proofs of the interactive consistency conditions rather than the actual definition of OMIC. The proof was complicated only slightly by the need for our extra function `OMIC1`.

Our formalization of the two conditions IC1 and IC2 are given in Figure 7. Notice that, following Rushby, we introduce the desired property as a universally quantified property, and then prove by induction a lemma stating that the property holds invariantly under appropriate conditions. Except for the notational difference and the fact that some type information is transferred from the function definitions to explicit hypotheses, the statements of Rushby's version and ours are essentially identical. As with Rushby's version, the final theorems on the full set of processes follows immediately from the inductive versions.

Following Rushby's proof structure[15], we were able to formulate and prove our versions of the interactive consistency conditions in a little over a day.[7] Except for replacing some quantified notions in Rushby's version by recursive functions, our proof structure is very similar to his. The complete ACL2 script of the proof is available from the author.

6 Comparisons

Different automated proof systems have their own strengths and weaknesses. ACL2 is an expressive logic closely tied to a real programming language. The prover is powerful and versatile, particularly with respect to problems couched in its "native" domain of recursively defined functions. Because of the higher-order functions and embedded lambda expressions Rushby's formulation of the Oral Messages algorithm problem was not an obvious candidate for treatment in ACL2. Nevertheless, we feel that ACL2 performed admirably, without any real complication in the statement of the algorithm or the proofs.

[7] We actually did the proof first in Nqthm and then redid it in ACL2. The ACL2 version is cleaner, but quite similar to the Nqthm version.

```
(defun-sk C1-PROP (r)
  (forall (p q y caucus v)
    (implies
     (and (ok p)
          (ok q)
          (proc-setp caucus)
          (member-equal p caucus)
          (member-equal q caucus)
          (member-equal y caucus)
          (< (* 3 r) (len caucus))
          (<= (fault-count caucus) r))
     (equal
       (omic r v caucus p y) (omic r v caucus q y)))))

(defthm C1-PROP-LEMMA
  (implies (and (<= r (m))
                (naturalp r))
           (c1-prop r)))

(defun-sk C2-PROP (r)
  (forall (p q caucus v)
    (implies
     (and (ok p)
          (ok q)
          (proc-setp caucus)
          (member-equal p caucus)
          (member-equal q caucus)
          (< (+ (* 2 (fault-count caucus)) r)
             (len caucus)))
     (equal
      (omic r v caucus) (binding q v)))))

(defthm C2-PROP-LEMMA
  (implies (<= r (m))
           (c2-prop r)))
```

Figure 7: The ACL2 Version of IC1 and IC2

6.1 Syntactic Issues

The Lisp syntax is a major stumbling block for many new users of ACL2. But it has the advantages of being completely unambiguous without any ancillary precedence rules, and is trivially machine parsed, making it much easier to perform meta-theoretic reasoning about terms in the logic, for example. "Traditional" notation requires considerably more effort to parse. Some recent research[6] suggests that so-called "standard mathematical notations" in fact may be quite difficult to understand except to the expert, exactly because of such complex ancillary rules. For example, we noted in [21] that the published proof of the Interactive Convergence Clock Synchronization algorithm in EHDM[18] is unnecessarily confusing in places because the expected precedence was not observed in generating "traditional" notation from the EHDM internal form.

In our translation of the OMBG version of the algorithm into ACL2 we encountered another instance of the ambiguity of "traditional" notation. In our initial translation we misinterpreted the following lambda expression:

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

This should be interpreted as

$$\lambda q : [\mathrm{OMBG}\,(q, r-1,\ \mathrm{send}\,(r, t, G, q),\ \mathit{caucus}\text{-}\{G\})(p)],$$

rather than as

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

In our initial attempt to translate the PVS version into ACL2, we made the wrong reading and this error took an hour or so to discover and correct. As an experiment, we added parentheses to the PVS script and tried to parse both renderings. Each parsed without complaint. Only the failure of Rushby's proof to replay for the mistaken interpretation distinguished the two versions. The PVS type correctness conditions might somehow catch the error, but we are not sure.

In fairness, our precedence error might have been immediately obvious to a seasoned PVS user or to an expert in lambda calculus. However, any but the most cautious reader might be led astray by an expression such as, $A\&B = B\&A$ and $A\&B <=> B\&A$, where A and B are Boolean variables. Because of the precedence rules of PVS the first would be parsed as: $(A\&(B = B)\&A)$, but the second as $(A\&B) <=> (B\&A)$.[8] The possibility of such anomalies points up one strong advantage of the unambiguous Lisp syntax.

The Lisp syntax of ACL2 derives from the decision to make a prover for a real programming language. Thus, in addition to analysis of abstract algorithms, ACL2 can verify efficiently executable implementations. An input capability currently under development should produce the benefit of allowing a more traditional syntax while preserving the core strengths of the ACL2 logic and prover and the tie to executable Common Lisp.

We agree with Rushby that "The value of syntactic conveniences [...] should not be underestimated; we find that they reduce learning time, and ease comprehension and communication" [12]. But they also should not be overestimated. Our experience is that such syntactic issues are a relatively minor concern for *serious* users of automated proof tools, when compared to the intellectual effort of analyzing complex algorithms. We acknowledge, however, that syntax does present an issue when publishing formal verifications and making them available to a wider audience. For that reason, many of our published papers (e.g. [23]) use a more traditional mathematical syntax similar to that displayed earlier.

6.2 Features of the Logic

Is our formalization of the OMIC algorithm as perspicuous as Rushby's, independent of the surface syntax of the logic? The answer depends on whom you ask. For an audience used to recursive programming idioms and uncomfortable with higher order logics, our version is probably much more perspicuous. The ability in EHDM and PVS to use strong typing, general lambda expressions, higher order functions, and unrestricted quantification undoubtedly does provide an expressive specification capability. However, the real utility of each of these features in the difficult process of analyzing complex algorithms is largely a matter of taste, one might even say a matter of "religion."

For example, the question of how strongly typed a programming and specification language should be is a matter of continuing debate. For example, consider the following opinion from Leslie Lamport:

[8] Thanks to one of the anonymous reviewers for pointing out this example to me.

> [T]ypes are not just unnecessary, they are harmful in formalizing mathematics...Most computer scientists have an unreasonable attachment to types...We can assure the reader that mathematicians have gotten along quite well for two thousand years without types, and they still can today[8].

Lisp derives much of its expressive utility from its weak typing; on the other hand, considerable execution efficiency can be gained from type declarations. ACL2 preserves both benefits by the judicious integration of an untyped logic with guards to insure compliance with efficiently executable raw Lisp implementations[11]. In theorems, "type" hypotheses can be included where needed. As in PVS this permits using arbitrary predicates as "types" but also requires arbitrarily hard theorem proving to assure type compliance.

Regarding higher order features and lambda expressions: they often are elegant, but seldom necessary. It is undoubtedly possible to contrive PVS specification which make essential use of the higher order aspects of the language and could not be translated into a first order context; however, these seem to be rare. We have found in translating several specifications (e.g. the Interactive Convergence Clock Synchronization Algorithm[21] and the current effort) from a higher order framework to a first order framework that we have not had any difficulty. This suggests that the "higher orderness" of these specification is primarily a notational convenience.

Most lambda terms can be translated directly into recursive functions, though often at the cost of additional parameters. Of course, lambda expressions returning lambda terms are inherently higher order. Use of such expressions and of higher order functions sometimes yields a terse and elegant representation, though not necessarily a particularly perspicuous one. It is not really clear how many potential users of a specification language are familiar and comfortable with higher order functions.

Finally, the limited style of quantification available in ACL2 often requires defining a separate quantified notion rather than using a quantifier in place. In many instances, we have found that a call to an appropriate recursive function serves our purposes better than a quantifier. For proving properties of such recursive functions, the ACL2 prover is outstanding. However, the availability of both quantification and recursion in ACL2 is often a boon. Full quantification undoubtedly would lead to cleaner specifications in some cases.

Rushby's main claim in comparing his proof to the Bevier/Young version is that the richness and expressiveness of the specification language at his disposal led him to discover cleaner and more perspicuous versions of the algorithms. Since it reports personal experience, this claim is irrefutable. However, the implied suggestion that most other potential users will be able to make truly effective use of higher order functions, lambda abstraction, etc. is dubious, as is the suggestion that the resulting specifications are likely to be more readible or more accessible.

6.3 Proof Issues

Simply by virtue of its richness, higher order logic can present challenges to automated proof discovery tools that are not encountered for first order logic. For example, unification is straightforward for first order logic, difficult for second order (producing perhaps an infinite number of unifiers), and undecidable for third order. Thus, the elegance gained in specification must be weighed against the potential additional proof burden. Perhaps this burden is necessary for problems which are inherently higher order. However, as we suggested above, such problems seem to be rare. Use of a more restrictive logic means that more effort sometimes must be invested in the specification, but is often repaid with increased automation in the proof.

Differences in the PVS and ACL2 provers arise from differences in the logics supported and in the goals of the designers. The PVS prover has the feel of an interactive proof-checker but with some powerful proof techniques and decision procedures built in. A PVS proof is an explicitly interactive activity which can be carried on at a fairly low level, making PVS somewhat more accessible to new users. Proofs are stored as an explicit tree of prover commands. ACL2 is more in the automated proof school, with guidance to the prover given in the form of a sequence of "events"–definitions and theorems which successively "educate" the prover. This knowledge is used in subsequent proofs under control of various heuristics. The proof of any theorem is implicit in the sequence of events leading up to it. However, an available interactive interface to ACL2 allows low-level proof checking as well.

The main hazard of low-level proofs in either system is that such proofs are fragile. Any upgrade to the specification or to the prover may cause an earlier proof script to fail to replay because some of the explicit low-level steps recorded in the script no longer apply or have been obviated. We encountered this in attempting to replay Rushby's PVS proofs of OMIC and OMBG. An upgrade to the prover after these proofs had been recorded had apparently caused the induction mechanism to automatically prove one of the cases which had previously required explicit action on the user's part. Consequently, the proofs would not replay without our understanding the rather cryptic proof scripting notation and manually editing the proof tree. Low-level ACL2 proofs are susceptible to the same problem; for that reason most accomplished ACL2 users avoid them. We find that the expense of developing appropriate definitions and general, powerful rewrite rules usually pays off in proof modification and replay. We see no reason why this lesson could not be applied in PVS as well. However, disciplined avoidance of low-level proof "hacking" would eliminate some of the apparent accessibility of the PVS proof-checker interface.

7 Conclusions

In serious applications of automated reasoning the difficulty of accommodating a more restrictive notation is swamped by the intellectual cost of understanding and formalizing a difficult problem, and proving something about it. We believe that an improved logical formalism would have had a marginal effect on the size and manageability of the models in any of the large applications we have tackled. We have found the versatility and proof power of ACL2 completely up to the task.

We agree with John Rushby that "comparisons [between verification systems] 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" [15]. However, it is very easy to overstate the merits of one system over another in such comparisons either by selecting idiosyncratic criteria of comparison or by comparing performance of systems on two very different formulations of a problem. For example, if the criteria of merit were the number of open questions in mathematics solved with the prover, the resolution-based Aura system[20] would win hands down. However, Aura likely would not perform well on verifying the Oral Messages algorithm. We believe strongly that the obvious improvement of Rushby's formalization and proof of the Oral Messages algorithm over the version by Bevier and Young is due to his greater insight into the problem, not to any features of EHDM or PVS.

Overly enthusiastic promotion of any automated reasoning system does damage to the entire field by raising expectations that cannot be fulfilled and may foster pervasive misperceptions which are difficult to correct. For example, in a recent report, Ricky Butler describes the translation of some informal requirements specifications for a simple autopilot into a formal model using PVS[5]. Butler makes the assertion that "the modeling techniques used in this paper can be used with other formal specification languages that *are based upon higher order logic*" (emphasis added). In fact, the reader will be hard pressed to find even one higher order feature in Butler's specification; the entire specification can easily be translated to a first order formalism such as ACL2.[9] To demonstrate that this is so, we translated Butler's specification into ACL2 in about two hours work [22]. For the one lemma for which Butler gave proof performance numbers, ACL2 running on a slower machine dispatched the lemma totally automatically in less than one tenth the time. Yet, the casual assertion that higher order logic is somehow required for this specification may unfairly bias the non-expert against even considering first order systems for similar specification problems.

We again agree with Rushby that:

> Intrinsically hard problems generally involve complex interactions, such as the coordination of distributed, concurrent, or real-time computations, and redundancy management. It requires great skill to address these problems using formal methods, but the number and size of these problems may not be large. The greatest return on formal methods may be obtained when relatively few, very highly skilled people apply formal methods to the hardest and most critical problems. [17]

Whereas two state-of-the-art general-purpose verification environments such as PVS and ACL2 have different strengths and weaknesses, we believe that either can be an effective and powerful tool in the hands of a skilled user. Both have been used for some very sophisticated specification and proof efforts. On the other hand, at the current stage of automated reasoning development,

[9]In a private correspondence, Butler reminded us that record structures in PVS are represented as functions. Therefore, there are higher order aspects hidden by the record syntax. However, these are easily handled in a first order context.

neither will be particularly useful in the hands of anyone unwilling to invest considerable intellectual effort.

Acknowledgements: We wish to acknowledge the help and support of our colleagues at Computational Logic, particularly Bob Boyer, J Moore, and Matt Kaufmann, who designed and built ACL2 and have been unfailing in supporting its use. Bill Bevier has been a splendid collaborator in much related work. Rich Cohen provided some very helpful comments on an earlier draft of the paper. The anonymous reviewers made some extremely insightful and helpful comments. We would also like to acknowledge John Rushby's technical inspiration; he has been a model of clear thinking and exposition over many years.

References

[1] W.R. Bevier and W.D. Young. The proof of correctness of a fault-tolerant circuit design. In *Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications*, pages 107–114. IFIP, February 1991.

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

[3] R.S. Boyer and J S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. Technical Report ICSCA-CMP-52, Institute for Computing Science, University of Texas at Austin, January 1987.

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

[5] R.W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. Technical Memorandum 110255, NASA, May 1996.

[6] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[7] M. Kaufmann and J S. Moore. Design goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[8] L. Lamport. Types considered harmful. DEC SRC Internal note, December 1992.

[9] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[10] J S. Moore. Mechanically verified hardware implementing an 8-bit parallel IO Byzantine agreement processor. Technical Report Technical Report 69, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August 1991.

[11] J S. Moore and M. Kaufmann. ACL2: An industrial strength version of Nqthm. *Transactions on Software Engineering*, 1997.

[12] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2), February 1995.

[13] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In D. Kapur, editor, *Eleventh Conference on Automated Deduction*, number 607 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

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

[15] J. Rushby. Formal verification of an oral messages algorithm for interactive consistency. Contractor Report NAS1-18969, NASA, October 1992.

[16] J. Rushby. Formal methods and their role in certification of critical systems. Technical Report CSL-95-1, SRI International, Computer Science Laboratory, March 1995.

[17] J. Rushby. Formal methods and their role in digital systems validation for airborne systems. Contractor Report 4673, NASA, August 1995.

[18] J. Rushby and F. von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, January 1989. Draft.

[19] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, SRI International, Computer Science Laboratory, February 1991.

[20] L. Wos and S. Winker. Open questions solved with the assistance of AURA. In *Automatic Theorem Proving: After 25 Years*. American Mathematical Society, 1984.

[21] W.D. Young. Verifying the interactive convergence clock synchronization algorithm using the Boyer-Moore theorem prover. Contractor Report 189649, NASA, April 1992.

[22] W.D. Young. Specification of a simple autopilot in ACL2. Internal note 327, Computational Logic, Inc., July 1996.

[23] W.D. Young and W.R. Bevier. Mathematical modeling and analysis of an external memory manager. Technical Report 105, Computational Logic, Inc., October 1994.