NASA/TM–2004–213278

Model Checking Failed Conjectures in Theorem Proving: A Case Study

Lee Pike, Paul Miner, and Wilfredo Torres-Pomales
Langley Research Center, Hampton, Virginia

October 2004

NASA/TM–2004–213278

Model Checking Failed Conjectures in Theorem Proving: A Case Study

Lee Pike, Paul Miner, and Wilfredo Torres-Pomales
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681–2199

October 2004

Acknowledgments

This research was supported, in part, by Research Cooperative Agreement No. NCC-1-02043 awarded to the National Institute of Aerospace while the first author was a visitor. Additional support came from NASA's Vehicle Systems Program. The PVS and SAL tools used were developed by SRI International. We used PVS theories developed by members of the NASA Langley Formal Methods Group and the National Institute of Aerospace; in particular, we thank Alfons Geser, Jeffrey Maddalon for their theory development. As noted, the SAL model was adopted from work by Rushby [1].

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076–1320
(301) 621–0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161–2171
(703) 605–6000

Abstract

Interactive mechanical theorem proving can provide high assurance of correct design, but it can also be a slow iterative process. Much time is spent determining why a proof of a conjecture is not forthcoming. In some cases, the conjecture is false and in others, the attempted proof is insufficient. In this case study, we use the SAL family of model checkers to generate a concrete counterexample to an unproven conjecture specified in the mechanical theorem prover, PVS. The focus of our case study is the ROBUS Interactive Consistency Protocol. We combine the use of a mechanical theorem prover and a model checker to expose a subtle flaw in the protocol that occurs under a particular scenario of faults and processor states. Uncovering the flaw allows us to mend the protocol and complete its general verification in PVS.

1 Introduction

Although rarely discussed, interactive mechanical theorem proving is inherently an iterative process. A specification and a set of requirements are modeled in a theorem prover, and then conjectures that the specification satisfies the requirements are posited. Many times, one is unable to prove a conjecture. Provided the theorem prover is sound (and the conjecture is not true but unprovable—a possibility in mathematics), there are two reasons for this. First, the conjecture may be true, but the user lacks the resources or insight to prove it. Second, the conjecture may not hold; i.e., the specification fails to satisfy the stated requirement. It can be difficult to determine which of these is the case.

When mathematicians cannot complete a proof of a conjecture, they begin to seek a counterexample to it. Mechanical theorem proving can exacerbate this difficult task [2]. This is due in part to the nature of the objects specified in these systems. Proofs of correctness for algorithms and protocols often involve nested case-analysis. A proof obligation that cannot be completed is often deep within the proof, where intuition about the system behavior—and what would constitute a counterexample—wanes. This is also due to the nature of mechanical theorem proving. The proof steps issued in such a system are fine-grained. Formal specifications often make explicit detail that is suppressed in informal models. The detail and formality of the specification and proof can make the discovery of a counterexample more difficult.

This paper describes a case study to exploit model checking to facilitate interactive mechanical theorem proving. We describe the formal verification of a distributed fault-tolerant protocol in the mechanical theorem prover PVS [3]. A conjecture about the protocol is partially verified by case-analysis, leaving a single unproven case. The case involves a complex set of fault statuses and system invariants. To determine whether this case in fact gives rise to a counterexample to the conjecture, we modeled the single proof obligation in the recently-developed Symbolic Analysis Laboratory (SAL) [4], a tool that includes the symbolic and bounded model checkers used in this study. A counterexample is generated in SAL, suggesting a fix to the protocol. A proof of correctness for the new protocol is then carried through in PVS.

The protocol investigated is an interactive consistency protocol for use in the Reliable Optical Bus (ROBUS), a state-of-the-art ultra-reliable communications bus under development at the NASA Langley Research Center and the National Institute of Aerospace. It is being developed as part of the Scalable Processor-Independent Design for Electromagnetic-Resilience (SPIDER) architectures [5, 6]. SPIDER is a family of ultra-reliable architectures built upon the ROBUS. Currently, ROBUS implementations include both FPGA-based and software-based prototypes. More sophisticated prototypes are under development.

The counterexample was initially discovered by one of the authors by "engineering insight." We believed the subtlety of the counterexample provided a good basis for this case study. The counterexample is of particular interest to practitioners of formal methods: we present a "second-order bug" in a fault-tolerant distributed protocol. The bug's existence depends on two Byzantine faults [7,8] to simultaneously occur in the system. The system is designed to tolerate such a fault scenario. However, the subtlety of the scenario likely means it would likely escape detection during fault-injection testing [9]. Safety-critical systems oftentimes must have a failure rate no higher than 10^{-9} to 10^{-12} per hour of operation [10–12]. A design error that escapes testing could adversely affect a system's reliability. We believe that if our engineer had not discovered the bug, we would have discovered it via the formal methods being used in the development process for SPIDER.

The bug arises from the interaction between the system's fault assumptions and the local diagnoses made by nodes in the system. Local diagnoses are used in a fault-tolerant system to increase reliability and to maintain *group membership*, a group of mutually-trusted non-faulty nodes [13]. In a sense, the bug is due to the interplay of system operation (i.e., executing the protocol) and system survival (i.e., maintaining group membership). These concerns apply to variety of fault-tolerant embedded systems [10].

Contributions We make the following contributions in this case study. First, we describe how to enhance the efficiency of interactive mechanical theorem proving by using model checking to extract counterexamples from incomplete proof obligations. Second, we describe an interesting design error arising from the complex relationship between local diagnostic information and faults in the system. This sort of error has the potential to arise in other protocols that make use of continuous diagnostic data, and its subtlety suggests the importance of formal methods for design assurance.

Organization We describe the ROBUS Interactive Consistency Protocol as well as the architecture on which it is intended to execute, in Sect. 2. In Sect. 3, we describe the kinds and number of faults under which the ROBUS IC Protocol should correctly execute. Section 4 states the correctness requirements for the protocol as well as the state invariants that must hold for the ROBUS IC Protocol to satisfy them. In Sect. 5, we informally describe the counterexample, discuss its origins, and provide a "fix" for it. In Sect. 6 we describe the conjecture attempted in PVS and then our generation of a counterexample using SAL. Related work and concluding remarks are in Sect. 7.

2 The ROBUS IC Protocol

After describing the architecture of the ROBUS, on which the ROBUS IC Protocol is designed to execute, we describe the behavior of the protocol itself.

Architecture The architecture of the ROBUS is a fully-connected bipartite graph of two sets of nodes, *Bus Interface Units* (BIUs) and *Redundancy Management Units* (RMUs). BIUs provide the interface between the bus and hosts running applications that communicate over the bus. The RMUs provide fault-tolerance redundancy. The architecture for the special case of three BIUs and three RMUs is shown in Fig. 1. There must be a minimum of one BIU and one RMU in the architecture.

Diagnostic Data Understanding the protocol behavior requires a preliminary understanding of the diagnostic data collected by nodes. The protocol has a greater chance of succeeding if good nodes ignore faulty ones. Consequently, nodes maintain *diagnoses*

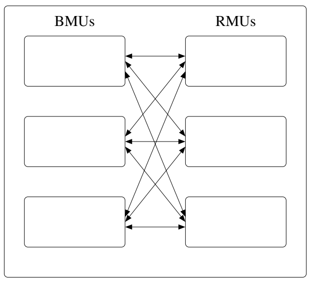

Figure 1. The ROBUS Architecture

against other nodes. These diagnoses result from mechanisms to monitor the messages received during protocol execution. Diagnostic data is accumulated over multiple protocol executions.

Each node maintains a *diagnostic function* assigning each node (including itself) to one of the following three classifications: *trusted*, *accused*, and *declared*. We call the node being labeled the *defendant*. Every (non-faulty) node assigns every other node to exactly one class. If a node labels a defendant as trusted, then the node has insufficient evidence that the defendant is faulty. If it labels a defendant as accused, then it has local evidence that the defendant is faulty, but does not know whether other good nodes have similar evidence. Once a defendant is declared, all good nodes know that they share the declaration.

Periodically, the RMUs and BIUs execute a *Distributed Diagnosis Protocol* in which the nodes submit the diagnoses accumulated thus far [13]. If enough good nodes have accused a defendant, then the defendant is *declared*. The Distributed Diagnosis Protocol ensures that all good nodes agree on which nodes have been declared.

2.1 Protocol Description

Distinguish one BIU as the General. The ROBUS IC Protocol is a synchronous protocol designed to reliably transmit the General's message despite faults in the system (the formal requirements are provided in Sect. 4). In the following, a *benign message* is one that all nonfaulty nodes can detect came from a faulty node (see Sect. 3). The ROBUS IC Protocol is as follows:

1. The General, G, broadcasts its message, v, to all RMUs.
2. For each RMU, if it receives a benign message from G, then it broadcasts the special message *source error* to all BIUs. Otherwise it relays the message it received.
3. For each BIU b, if b has declared G, then b outputs the special message *source error*. Otherwise, if i received a benign message from an RMU, then that RMU is accused. b performs a majority vote over the values received from those RMUs it trusts. If no majority exists, *source error* is outputted; otherwise, the majority value is outputted.

3 Faults

Fault Classifications Faults result from innumerable occurrences including physical damage, electromagnetic interference, and "slightly-out-of-spec" communication [7]. We collect these fault occurrences into *fault types* according to their effects in the system.

We adopt the *hybrid fault model* of Thambidurai and Park [14]. All non-faulty nodes are also said to be *good*. A node is called *benign*, or *manifest*, if it sends only *benign messages*.

Benign messages abstract various sorts of misbehavior. A message that is sufficiently garbled during transmission may be caught by an error-checking code and deemed benign. In synchronized systems with global communication schedules, they also abstract messages not sent (i.e., a message is expected by a receiver but is absent on a communication channel) at unscheduled times. A node is called *symmetric* if it sends every receiver the same message, but these messages may be incorrect. A node is called *asymmetric*, or *Byzantine* [15], if it arbitrarily sends different messages to different receivers.

The Fault Assumptions A fault-tolerant protocol is designed to tolerate a certain number of faults of each kind of fault type. For a protocol, this is specified by its *maximum fault assumption* (MFA). A proof of correctness of a protocol is of the form, "If the MFA holds, then the protocol satisfies property P," where P is a correctness condition for the protocol. The probability that a MFA holds is determined by reliability analysis [16].

We call the MFA for the ROBUS IC Protocol the *Interactive Consistency Dynamic Maximum Fault Assumption* (IC DMFA). 'Dynamic' emphasizes that the fault assumption is parameterized by the local diagnoses of nodes, which change over time.

Definition 1 (IC DMFA). Let *GB*, *SB*, and *AB* denote the sets of BIUs that are good, symmetrically-faulty, and asymmetrically-faulty, respectively. Let *GR*, *SR*, and *AR* represent the corresponding sets of RMUs, respectively. For good BIU b, let $\mathtt{T}_b$ denote the set of RMUs b trusts. This is b's *trusted set.* Define $\mathtt{T}_r$ similarly—it is the set of BIUs that RMU r trusts. The following formulas together make up the IC DMFA. G is the General. For all BIUs b and RMUs r,

1. $|GR \cap \mathtt{T}_b| > |SR \cap \mathtt{T}_b| + |AR \cap \mathtt{T}_b|$;
2. $G \in AB \cap \mathtt{T}_r$ implies $|AR \cap \mathtt{T}_b| = 0$.

The first clause ensures that a good BIU b contains strictly more good RMUs in $\mathtt{T}_b$ than it does symmetrically-faulty or asymmetrically-faulty RMUs. The second clause ensures that either no good RMU r trusts an asymmetrically-faulty General, or no good BIU b trusts an asymmetrically-faulty RMU.

4 The ROBUS IC Protocol Correctness

We begin by stating the requirements for the ROBUS IC Protocol. We then state invariants that must hold in a system executing the ROBUS IC Protocol in order for it to meet these requirements.

Requirements Two requirements must hold.

Definition 2 (Agreement). All good BIUs compute the same value.

Definition 3 (Validity). If the General is good and broadcasts message v, then the value computed by a good BIU is v.

Diagnostic Assumptions In addition to constraining the number of and kind of faults, the correctness of the ROBUS IC Protocol depends on the diagnostic mechanisms satisfying certain constraints. Let b_1 and b_2 be good BIUs, and let n be either a BIU or RMU of any fault classification.

Definition 4 (Good Trusted). b_1 trusts n if n is *good.*

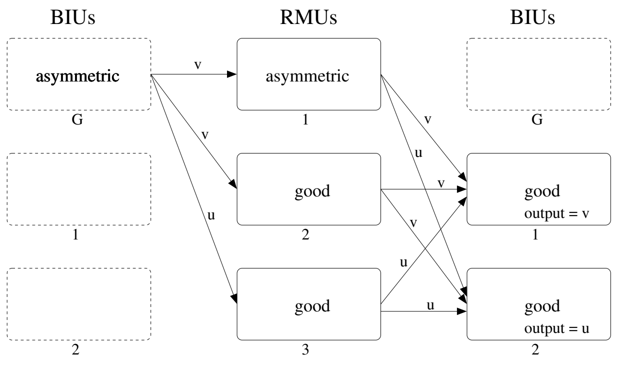

Figure 2. An Instance of the ROBUS IC Protocol Violating Agreement

Definition 5 (Symmetric Agreement). If n is not asymmetrically-faulty, b_1 accuses n if and only if b_2 accuses n.

Definition 6 (Conviction Agreement). b_1 declares n if and only if b_2 declares n.

These properties similarly hold for any two good RMUs with respect to a defendant n. Conviction Agreement also holds between good BIUs and good RMUs.

Intuitively, *Good Trusted* ensures that diagnostic mechanisms never lead a good node to accuse another good node. *Symmetric Agreement* ensures that all good nodes that receive the same data make the same diagnosis. Note, however, that Symmetric Agreement allows a good BIU and a good RMU to make different diagnoses about a node that is asymmetrically-faulty. Finally, *Conviction Agreement* is a correctness requirement of the Distributed Diagnosis Protocol [13], and it is a precondition for the correctness of the protocol under investigation in this paper. Together, these three assumptions are called the *Diagnostic Assumptions*.

5 The Counterexample

We describe the counterexample informally and briefly describe its origins. We then describe a protocol that does not suffer from the flaw.

A Counterexample Instance The following instance of the ROBUS IC Protocol violates Agreement. Consider an architecture containing three BIUs, G, b_1, and b_2, and three RMUs r_1, r_2, and r_3. Let the General be asymmetrically-faulty. Let RMU r_1 be asymmetrically-faulty, too, and let all other nodes be good. Suppose b_1 and b_2 either accuse or trust G (it does not matter which), and they trust all RMUs. Furthermore, suppose b_1 and b_2 trust r_1, but no good RMU trusts G. These hypotheses satisfy the IC DMFA and the Diagnostic Assumptions. Agreement is violated if the following instance of the ROBUS IC Protocol transpires, as illustrated in Fig. 2.

1. G sends message v to r_1 and r_2, and it sends message u to r_3, where $v \neq u$.

2. r_1 sends message v to b_1 and u to b_2. r_2 sends message v to both b_1 and b_2. r_3 sends message u to both b_1 and b_2.

3. b_1 outputs v whereas b_2 outputs u.

Origins of the Flaw The flaw in the ROBUS IC Protocol was introduced when an earlier version of the protocol was amended to allow for the reintegration of transiently-faulty nodes. A node becomes transiently-faulty when its state is disrupted (due, e.g., to exposure to high-intensity radiation), but the node is not permanently damaged. A node that suffers a transient fault has the potential to *reintegrate* with the good nodes in the system by restoring consistent state with them.

In the earlier version of the ROBUS IC Protocol, an RMU would only relay a message from the General if it trusted the General. Otherwise, the *source error* message was relayed. To allow for reintegration, the messages from a previously-declared General needed to be relayed by RMUs so that the BIUs can determine whether it is fit for reintegration. However, the flaw in the ROBUS IC Protocol arose when the earlier protocol was changed so that RMUs relayed the message from the General regardless of its diagnostic status, so long as it did not send a benign message.

A New ROBUS IC Protocol In retrospect, a fix to the protocol is simple. Step 2 of the protocol description in Sect. 2.1 is changed so that an RMU r relays the message *source error* if it receives a benign message or if r accuses the General. If the General is declared, its message is relayed to allow BIUs to gather diagnostic data about the General. An accused General implies that the General recently suffered a fault (assuming the accuser is good), so there is no need to relay its message for reintegration purposes. The correctness of the protocol, proved in PVS, is described in [17].

6 Formally Deriving the Counterexample

In this section, we describe the unfinished proof obligation generated in our attempt to formally prove a conjecture about the ROBUS IC Protocol. We then describe our use of a model checker to derive a counterexample to the conjecture.

6.1 Generating the Proof Obligation

In this study, we use the PVS theorem proving system developed by SRI International [3]. We have used PVS to specify and verify other ROBUS protocols [13, 17]. The specification language of PVS is a strongly-typed higher-order logic. The proof system is the classical sequent calculus.

Various details about the construction of the underlying theories used to model the algorithm and ROBUS are irrelevant.[1] A discussion of the abstractions used in this verification project can be found in [18], and a description of the theories developed for this verification can be found at [13, 19]. A discussion of how to specify an Oral Messages protocol as a higher-order recursive function can be found in [20]; our more complex model is in the same spirit (additional complexity is introduced by specifying a hybrid fault model and the IC DMFA, which requires modeling local diagnoses). PVS files are available on-line [21]. The following notation is used in the formal statements of the Agreement Conjecture and the unproved sequent.

Variables and Parameters Let `B` and `R` be natural numbers. The set of BIUs and RMUs are indexed from 0 to `B - 1` and 0 to `R - 1`, respectively. These sets of indicies are denoted `below(B)` and `below(R)`, respectively. Let `b1`, `b2`, `G` $\in$ `below(B)`, where `G` is used to designate the General. `F` is a higher-order diagnostic function such that `F‘BR` denotes the collection of the BIUs' diagnoses against the RMUs, `F‘BB` denotes the BIUs'

[1]The PVS models were designed to model fault-tolerant protocols other than the ROBUS IC Protocol.

```
Agreement: CONJECTURE
    good?(b_status(b1)) AND
    good?(b_status(b2)) AND
    all_correct_accs?(b_status, r_status, F) AND
    IC_DMFA(b_status, r_status, F)
  =>
    robus_ic(b_status, r_status, F`BB(b1)(G), F`RB(b1))
            (G, msg, b1) =
    robus_ic(b_status, r_status, F`BB(b2)(G), F`RB(b2))
            (G, msg, b2)
```

Figure 3. The Agreement Conjecture

diagnoses against the BIUs, and similarly for `F‘RB` and `F‘RR`. `F‘RB(b1)(r)` denotes `b1`'s diagnosis of `r`, and similarly for the other functions. `F‘RB(b1)(r)` yields a value from the set {`trusted`, `accused`, `declared`}. The function `b_status` is a function mapping BIUs to some fault class—one of `good`, `benign`, `symmetric`, and `asymmetric`, and similarly, `r_status` maps RMUs to a fault class.

Constants The following functions and relations are used:

- `good?` is a predicate that takes the fault status of a node and is true if the status is *good*. `benign?`, `symmetric?`, and `asymmetric?` are similarly defined.
- `all_correct_accs?` is a predicate formally stating the Diagnostic Assumptions defined in Sect. 4.
- `declared?` is a predicate that takes the diagnosis made by one node against a defendant node and is true if the defendant is declared. Similarly, `trusted?` is true if the defendant is trusted.
- `IC_DMFA` is a formal statement of the IC DMFA described in Sect. 3.
- `robus_ic` is a higher-order function that functionally models the ROBUS IC Protocol, as described in Sect. 2.1. It takes as arguments the fault statuses of the BIUs and RMUs, the diagnoses a BIU makes of `G`, as well as the set of its other diagnoses. It returns another function that takes the General's identifier, the message it sends, and a BIU identifier. The function returns the message the BIU outputs after the execution of the ROBUS IC Protocol.

The conjecture to be proved is stated in Fig. 3. Assuming that `b1` and `b2` are both good, that the Diagnoses Assumptions hold, and that the IC DMFA holds, we attempt to prove that the result of `robus_ic` is the same when applied to `b1` and `b2`.

Every branch of the conjecture in Fig. 3 is discharged except for the branch ending in the single sequent in Fig. 4 (irrelevant formulas have been omitted). PVS labels the formulas in the antecedent with negative integers, while those in the consequent are labeled with positive integers. It is also the convention of PVS to denote skolem constants with a trailing "`!n`," where `n` is some integer.

6.2 Model Checking the Sequent

We use the Symbolic Analysis Laboratory (SAL) [4,22], also developed by SRI International, to model check the protocol against the undischarged sequent. SAL is a family of state-of-the-art model checkers that includes symbolic, bounded, and explicit-state model checkers,

```
[-1]  good?(r_status!1(r!1))
[-2]  asymmetric?(b_status!1(G!1))
[-3]  IC_DMFA(b_status!1, r_status!1, F!1)
[-4]  all_correct_accs?(b_status!1, r_status!1, F!1)
  |-------
[1]   trusted?(F!1`BR(r!1)(G!1))
[2]   declared?(F!1`BB(b2!1)(G!1))
{3}   (FORALL (p_1: below(R)):
         (trusted?(F!1`RB(b1!1)(p_1)) =>
            NOT asymmetric?(r_status!1(p_1))))
       &
       (FORALL (p_1: below(R)):
          (trusted?(F!1`RB(b2!1)(p_1)) =>
             NOT asymmetric?(r_status!1(p_1))))
[4]   declared?(F!1`BB(b1!1)(G!1))
[5]   robus_ic(b_status!1, r_status!1,
               F!1`BB(b1!1)(G!1), F!1`RB(b1!1))
              (G!1, msg!1, b1!1)
       =
       robus_ic(b_status!1, r_status!1,
                F!1`BB(b2!1)(G!1), F!1`RB(b2!1))
               (G!1, msg!1, b2!1)
```

Figure 4. The Unproven PVS Sequent

among other tools. The SAL language includes many high-level constructs such as recursive function definition, synchronous and asynchronous composition operators, and quantifiers. We particularly exploit the quantifier, recursive function, and synchronous composition constructs.

Our SAL model builds on the model of Oral Messages that is explained in detail in Rushby's SAL tutorial [1]. Our model differs as we must represent the local diagnoses data of each node, the Diagnosis Assumptions, and the IC DMFA, which is parametrized by the local diagnoses. Furthermore, we state these constraints explicitly rather than embedding them into the system model. We found this makes our model more perspicuous. The model is available on-line [21].

To formulate the unproven sequent in the model checking logic LTL, we use the fact that a sequent can be read as stating that if the conjunction of the antecedent statements is true, then the disjunction of the consequent statements is true. That is, if $\mathcal{A}$ is the set of antecedents and $\mathcal{C}$ is the set of consequents, a sequent is equivalent to the conditional

$$\bigwedge \mathcal{A} \Longrightarrow \bigvee \mathcal{C} \, . \tag{1}$$

This formulation is used to express the sequent in SAL and appears in Fig. 5. There, `SYSTEM` denotes the model of the ROBUS IC Protocol developed in the model checker, the symbol `|-` denotes the purported satisfaction relation between the model and `G` is the global-state operator of LTL (not to be confused with the denotation of the General).

SAL has an imperative language, so some of the predicates in the PVS sequent have been expressed equationally. Some of the functions of PVS have been converted to arrays in SAL, giving rise to the bracket notation.

Two additional statements in the LTL formulation are artifacts of how the protocol is modeled in the model checker. First, there is a program counter `pc` that represents which round of the protocol is currently executing. These rounds correspond to the three rounds described in Sect. 2.1. When `pc = 4`, the last round has completed. The second artifact is the imperative definition of the result of the ROBUS IC Protocol using the array called

```
counterex: THEOREM SYSTEM |-
  G(  (pc = 4 AND
       r_status[1] = good AND
       G_status = asymmetric AND
       IC_DMFA(r_status, F_RB, F_BR, G_status) AND
       all_correct_accs(r_status, F_RB,
                        G_status, F_BR, F_BB))
    =>
      (F_BR[1] = trusted OR
       F_BB[2] = declared OR
       (FORALL (r: RMUs): F_RB[1][r] = trusted =>
          r_status[r] /= asymmetric AND
        FORALL (r: RMUs): F_RB[2][r] = trusted =>
          r_status[r] /= asymmetric) OR
       F_BB[1] = declared OR
       robus_ic[1] = robus_ic[2]));
```

Figure 5. The SAL Formulation of the Undischarged Sequent

robus_ic.

Thus, the conjecture in Fig. 5 can be read as stating that in every state reachable from the initial state of SYSTEM, the formulation of the unproven sequent described above is true.

A counterexample to the formula in Fig. 5 is a reachable state in which the formula is false. As mentioned, that formula is derived from the conditional interpretation of a sequent in (1). The negation of (1) is equivalent to

$$\bigwedge(\mathcal{A} \cup \bar{\mathcal{C}}) \;, \tag{2}$$

where $\bar{\mathcal{C}}$ denotes the negation of each formula in $\mathcal{C}$. A counterexample is therefore a reachable state in which (2) is true. In this state, all the antecedents are true, and every consequent is false, matching the informal description of the counterexample in Sect. 5.

We used SAL's symbolic model checker with no optimizing command-line arguments on a system with one gigabyte of memory and an AMD Athlon 2000+ processor. A counterexample like the one described in Sect. 5 was discovered in about 16 seconds for three RMUs and three BIUs, including the General.

One may wonder whether this counterexample arises from the system having too few RMUs to relay messages. Increasing the number of RMUs quickly overwhelms the symbolic model checker. However, we obtain a similar counterexample using SAL's bounded model checker for seven RMUs in a little over two minutes on the same system.

These concrete counterexamples demonstrate that the unproved sequent cannot be discharged because the protocol itself has an error. Changing the PVS and SAL models to include the fix suggested in Sect. 5 allows the Agreement proof to be completed (see [17]), and SAL verifies the formula in Fig. 5 in a sufficiently small model (the fix is included as commented code in the SAL model available on-line [21]).

6.3 Remarks on the Approach

In our case study, we manually modeled the protocol and the requirements, both in PVS and SAL. This was simultaneously advantageous and disadvantageous. Having to model the protocol and requirements in distinct languages provided an additional guard against modeling errors in each language. Such mistakes are easy to make; in particular, we found it was easy to generate a false negative in SAL (i.e., return no counterexample when the actual protocol does not satisfy the actual requirement). For example, we initially omitted the second universal quantifer in the third disjunct of the consequent of the LTL formula in Fig. 5:

```
FORALL (r: RMUs):
  F_RB[1][r] = trusted => r_status[r] /= asymmetric AND
  F_RB[2][r] = trusted => r_status[r] /= asymmetric
```

No counterexample was discovered. This is because conjunction evidently binds tighter than implication in SAL, changing the meaning of the formula. Had we only employed a model checker to check the protocol against the requirements (and had the counterexample not already be known), it might have been overlooked. This is less of a danger in a theorem prover. Due to their interactive nature, false negatives are harder to produce in a mechanical theorem prover (that is sound). A formal proof in a theorem prover can be reviewed for correctness, but a "model checking proof" cannot.

A disadvantage is the additional work required to model the protocol and requirements in two tools. Additionally, had the unproved proof obligation been the result of an erroneous PVS model, it may not have appeared in the SAL model. Of course, finding no counterexample in SAL would have led us to reexamine the model in PVS. SRI International has stated that future work includes developing interpreters from SAL to PVS [22]. Once implemented, one will have the choice of specifying a system and its requirements in both tools manually or to use the interpreter.

Some limitations of this approach to generate counterexamples to unproven proof obligations are inherent to the limitations of model checking in general. A model checker is useful when the system can be modeled as a state machine, and the requirements to be proved can be modeled in a temporal logic. Mechanical theorem provers are routinely used to specify and verify mathematical objects that do not lend themselves to these restrictions. As well, a counterexample may exist, but be beyond the computational limits of the model checker and the computer on which it is hosted. The power of the SAL model checkers and the expressiveness of its language (particularly, its synchronous operators), made this work feasible.

7 Conclusions and Related Work

We have described a case study of our use of mechanical theorem proving and model checking to turn a failed proof into a concrete counterexample revealing a substantial and interesting bug.

Protocols like the one described in this paper are fault-tolerant consensus algorithms and are known as "interactive consistency" or "oral messages" protocols. The protocol presented here is based on a protocol designed by Davies and Wakerly [17, 23]. Lynch's textbook provides a modern introduction to these sort of protocols as well as pointers into the literature [24]. Many of these protocols have been formally verified, both by theorem proving [25–27] and by model checking [1].

Mechanical theorem proving and model checking have been combined in a number of studies [2]. Most of these studies have focused on either using theorem proving in abstracting systems for model checking or using model checking to facilitate theorem proving. In fact, PVS has an embedded model checker that can be used to model check state-machine specifications with requirements stated in the mu-calculus. In [28], Havelund and Shankar develop a methodology to use theorem proving to derive a finite abstraction of a protocol that can be model checked for correctness.

As far as we know, little work has been done to use model checking to understand failed proofs in a mechanical theorem prover. Research applying other techniques to "non-theorems" can be found in [29,30]. Most related to this case study is [31], in which resolutionbased theorem proving and model checking are used to discover counterexamples to proof obligations. Our work differs in that we present a reasonably intricate protocol for an actively-developed system (a small illustrative example is presented in [31]). As well, the focus therein is on automated theorem proving; our focus is on using model checking to facilitate interactive theorem proving.

Practitioners have long desired better tool integration, but have faced a number of obstacles [32]. Our work is made possible by the expressiveness and power of the PVS and SAL tools, but as mentioned in Sect. 6.3, obstacles to complete integration remain.

References

1. Rushby, J.: SAL Tutorial: Analyzing the Fault-Tolerant Algorithm OM(1). CSL Technical Note, SRI International, 2004. Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`.

2. Rushby, J.: Integrated Formal Verification: Using Model Checking With Automated Abstraction, Invariant Generation, and Theorem Proving. *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops*, D. Dams, R. Gerth, S. Leue, and M. Massink, eds., Springer-Verlag, Trento, Italy, and Toulouse, France, vol. 1680 of *Lecture Notes in Computer Science*, July/Sept 1999. Available at `http://www.csl.sri.com/papers/spin99/`.

3. Owre, S.; Rusby, J.; Shankar, N.; and von Henke, F.: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, vol. 21, no. 2, February 1995, pp. 107–125.

4. SRI International: Symbolic Analysis Laboratory SAL. 2004. Available at `http://sal.csl.sri.com/`.

5. NASA Formal Methods Group: SPIDER Homepage. Website, 2004. Available at `http://shemesh.larc.nasa.gov/fm/spider/`.

6. Miner, P. S.; Malekpour, M.; and Torres, W.: Conceptual Design of a Reliable Optical Bus (ROBUS). *21st AIAA/IEEE Digital Avionics Systems Conference DASC*, Irvine, CA, October, 2002.

7. Driscoll, K.; Hall, B.; Sivencrona, H.; and Zumsteg, P.: Byzantine Fault Tolerance, from Theory to Reality. *Computer Safety, Reliability, and Security*, G. Goos, J. Hartmanis, and J. van Leeuwen, eds., Lecture Notes in Computer Science, The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP, Springer-Verlag Heidelberg, September 2003, pp. 235–248.

8. Pease, M.; Shostak, R.; and Lamport, L.: Reaching Agreement in the Presence of Faults. *Journal of of the ACM*, vol. 27, no. 2, 1980, pp. 228–234.

9. Hsueh, M.-C.; Tsai, T. K.; and Iyer, R. K.: Fault Injection Techniques and Tools. *IEEE Computer*, vol. 30, no. 4, 1997, pp. 75–82. Available at `citeseer.ist.psu.edu/hsueh97fault.html`.

10. Rushby, J.: Bus Architectures For Safety-Critical Embedded Systems. *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, T. Henzinger and C. Kirsch, eds., Springer-Verlag, Lake Tahoe, CA, vol. 2211 of *Lecture Notes in Computer Science*, Oct. 2001, pp. 306–323.

11. Koptez, H.: *Real-Time Systems*. Kluwer Academic Publishers, 1997.

12. Littlewood, B.; and Strigini, L.: Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, November 1993, pp. 69–80.

13. Geser, A.; and Miner, P.: A Formal Correctness Proof of the SPIDER Diagnosis Protocol. NASA/CP-2002-211736, NASA Langley Research Center, Hampton, Virginia, August 2002. Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLSs).

14. Thambidurai, P.; and Park, Y.-K.: Interactive Consistency With Multiple Failure Modes. *7th Reliable Distributed Systems Symposium*, October 1988, pp. 93–100.

15. Lamport; Shostak; and Pease: The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, vol. 4, July 1982, pp. 382–401. Available at `http://citeseer.nj.nec.com/lamport82byzantine.html`.

16. Butler, R. W.: The SURE Approach to Reliability Analysis. *IEEE Transactions on Reliability*, vol. 41, no. 2, June 1992, pp. 210–218.

17. Miner, P.; Geser, A.; Pike, L.; and Maddalon, J.: A Unified Fault-Tolerance Protocol. *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, Y. Lakhnech and S. Yovine, eds., Springer, vol. 3253 of *Lecture Notes in Computer Science*, 2004, pp. 167–182.

18. Pike, L.; Maddalon, J.; Miner, P.; and Geser, A.: Abstractions for Fault-Tolerant Distributed System Verification. *Theorem Proving in Higher Order Logics (TPHOLs)*, K. Slind, A. Bunker, and G. Gopalakrishnan, eds., Springer, vol. 3223 of *Lecture Notes in Computer Science*, 2004, pp. 257–270.

19. Miner, P.; and Geser, A.: A New On-Line Diagnosis Protocol for the SPIDER Family of Byzantine Fault Tolerant Architectures., April 2003. Available at `http://shemesh.larc.nasa.gov/fm/spider/spider_pubs.html`.

20. Rushby, J.: Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, vol. 25, no. 5, September 1999, pp. 651–660. Available at `http://www.csl.sri.com/papers/tse99/`.

21. NASA Formal Methods Group: PVS Proof Files. Website, 2004. Available at `http://shemesh.larc.nasa.gov/fm/spider/counterex/`.

22. SRI Computer Science Laboratory: Formal Methods Roadmap: PVS, ICS, and SAL. SRI-CSL-03-05, SRI International, Menlo Park, CA 94025, November 2003.

23. Davies, D.; and Wakerly, J. F.: Synchronization and Matching in Redundant Systems. *IEEE Transactions on Computers*, vol. 27, no. 6, June 1978, pp. 531–539.

24. Lynch, N. A.: *Distributed Algorithms*. Morgan Kaufmann, 1996.

25. Young, W. D.: Comparing Verification Systems: Interactive Consistency in ACL2. *IEEE Transactions on Software Engineering*, vol. 23, no. 4, April 1997, pp. 214–223.

26. Bevier, W.; and Young, W.: The proof of correctness of a fault-tolerant circuit design. *Second IFIP Conference on Dependable Computing For Critical Applications*, 1991. Available at `citeseer.ist.psu.edu/bevier91proof.html`.

27. Lincoln, P.; and Rushby, J.: Formal Verification of an Interactive Consistency Algorithm for the Draper FTP Architecture Under a Hybrid Fault Model. *Compass '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, IEEE Washington Section, Gaithersburg, MD, June 1994, pp. 107–120. Available at `http://www.csl.sri.com/papers/compass94/`.

28. Havelund, K.; and Shankar, N.: Experiements in Theorem Proving and Model Checking for Protocol Verification. *Proceedings of Formal Methods Europe FME'96*, Lecture Notes in Computer Science, Springer, 1996.

29. Steel, G.; Bundy, A.; and Denney, E.: Finding Counterexamples to Inductive Conjectures and Discovering Security Protocol Attacks. *Foundations of Computer Security*, I. Cervesato, ed., DIKU Technical Report, Copenhagen, Denmark, 25–26 July 2002, pp. 49–58. Available at `homepages.inf.ed.ac.uk/s9808756/papers/`.

30. Ahrendt, W.; Baumgartner, P.; and de Nivelle, H., eds.: *Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability*. Second International Joint Conference on Automated Reasoning, July 2004. Available at `http://www.cs.chalmers.se/~ahrendt/ijcar-ws-disproving/`.

31. Bicarregui, J. C.; and Matthews, B. M.: Proof and Refutation in Formal Software Development. *3rd Irish Workshop on Formal Methods (IWFM'99)*, July 1999.

32. Johnson, S. D.: View from the Fringe of the Fringe. *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, T. Margaria and T. Melham, eds., Springer-Verlag, vol. 2144 of *Lecture Notes in Computer Science*, 2001, pp. 1–12.

REPORT DOCUMENTATION PAGE					*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-10-2004	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE Model Checking Failed Conjectures in Theorem Proving: A Case Study		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Pike, Lee; Miner, Paul; Torres-Pomales, Wilfredo		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER 23-762-65-AD
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-2199		**8. PERFORMING ORGANIZATION REPORT NUMBER** L–18390
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001		**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM–2004–213278

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 64
Availability: NASA CASI (301) 621-0390 Distribution: Nonstandard

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://techreports.larc.nasa.gov/ltrs/ or http://ntrs.nasa.gov.

14. ABSTRACT
Interactive mechanical theorem proving can provide high assurance of correct design, but it can also be a slow iterative process. Much time is spent determining why a proof of a conjecture is not forthcoming. In some cases, the conjecture is false and in others, the attempted proof is insufficient. In this case study, we use the SAL family of model checkers to generate a concrete counterexample to an unproven conjecture specified in the mechanical theorem prover, PVS. The focus of our case study is the ROBUS Interactive Consistency Protocol. We combine the use of a mechanical theorem prover and a model checker to expose a subtle flaw in the protocol that occurs under a particular scenario of faults and processor states. Uncovering the flaw allows us to mend the protocol and complete its general verification in PVS.

15. SUBJECT TERMS
formal methods, theorem proving, model checking, tool integration, byzantine faults, distributed consensus, fault-tolerance

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: `help@sti.nasa.gov`)
U	U	U	UU	18	**19b. TELEPHONE NUMBER** ***(Include area code)*** (301) 621-0390