Diagnosing a Failed Proof in Fault-Tolerance: A Disproving Challenge Problem[⋆⋆⋆]

Lee Pike[1], Paul Miner[2], and Wilfredo Torres-Pomales[2]

[1] Galois Connections
Beaverton, Oregon, USA
leepike@galois.com
[2] NASA Langley Research Center
Hampton, VA, USA
{p.s.miner, w.torres-pomales}@larc.nasa.gov

Abstract. This paper proposes a challenge problem in disproving. We describe a fault-tolerant distributed protocol designed at NASA for use in a fly-by-wire system for next-generation commercial aircraft. An early design of the protocol contains a subtle bug that is highly unlikely to be caught in fault-injection testing. We describe a failed proof of the protocol's correctness in a mechanical theorem prover (PVS) with a complex unfinished proof conjecture. We use a model checking suite (SAL) to generate a concrete counterexample to the unproven conjecture to demonstrate the existence of a bug. However, we argue that the effort required in our approach is too high and propose what conditions a better solution would satisfy. We carefully describe the protocol and bug to provide a challenging but feasible case study for disproving research.

1 Introduction

Although rarely discussed in the archival literature, many attempts to prove conjectures using interactive mechanical theorem proving fail. Provided the theorem prover is sound and the conjecture is not both true and unprovable – a possibility in mathematics – there are two possible reasons for a failed proof attempt. First, the conjecture may be true, but the user lacks the resources or insight to prove it. Second, the conjecture may be false. 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. The difficulty is partly due to theorem provers often being used to reason about algorithms and protocols. Proofs of correctness in this domain 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. The difficulty is also due to the nature of mechanical theorem proving. The proof steps issued in such a system are fine-grained. Formal specifications make explicit much of the detail that is suppressed in informal models. The detail and formality of the specification and proof makes the discovery of a counterexample more difficult.

We present a case study that highlights this difficulty. We describe the formal verification of a distributed fault-tolerant protocol in the mechanical theorem prover PVS [2]. 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.

In particular, 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 Extended Reliability (SPIDER) project [3, 4]. SPIDER is a family of ultra-reliable architectures built upon the ROBUS. Currently, ROBUS implementations include a FPGA-based prototype.

The counterexample was initially discovered by the third author through "engineering insight." The bug in the protocol design occurs only when there are two simultaneous Byzantine faults [5, 6]. As described in greater detail later in the paper, 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 [7]. In a sense,

[⋆] 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. This paper is a revised extended abstract of an (unrefereed) NASA technical memorandum [1].
[⋆⋆] In the *Proceedings of DISPROVING'06: Non-Theorems, Non-Validity, Non-Provability*, 2006, pages 24–33.

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 [8].

The protocol is designed to tolerate such a fault scenario. However, the subtlety of the scenario means it is extremely unlikely the bug would be caught during fault-injection testing [9]. Nevertheless, safety-critical systems like SPIDER that are designed for use in commercial aircraft must have a failure rate no higher than 10^{-9} to 10^{-12} per hour of operation [8, 10]. A design error that escapes testing could adversely affect a system's reliability. We believe that if the bug had not been caught by insightful inspection, the only other way it would be caught is through formal analysis.

In the paper, we describe our approach to formally uncover the bug.[3] Specifically, we model the failed proof obligation in a model checker, and attempt to prove it holds in a model in which parameters have been interpreted with small constants. Using the counterexample generated by the model checker, one can quickly determine that the protocol is incorrect. Furthermore, the counterexample suggests the appropriate modification to correct the bug.

Motivation Unfortunately, we feel our approach is inadequate for the following reasons:

– The approach is too interactive and onerous. It requires manually specifying the protocol and failed conjecture in a model checker and manually correcting the specification in the theorem prover.
– The approach depends on the counterexample arising by instantiating the parameters with small finite values.
– Indeed, we would like a more automated approach to verify the parameterized protocol specification in the first place than is possible using mechanical theorem proving alone.

Therefore, we offer this case study as a challenge problem to the disproving community. We believe researchers will find this problem of interest for the following reasons:

– The protocol is industrially-relevant, and the bug is genuine.
– The protocol can be described in English in just a few paragraphs (as is done in this paper), but the behavior of the protocol itself is subtle.
– While we believe our approach is unsatisfactory, we also believe it approximates the best contemporary approach for disproving problems like the one presented (a purpose of this paper is to solicit evidence to the contrary).
– Formal specifications of the protocol in both a mechanical theorem prover and a model checker accompany this paper for reference.

Organization We describe the ROBUS Interactive Consistency (IC) Protocol as well as the architecture on which it is intended to execute in Section 2. In Section 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 Section 5, we informally describe the counterexample, discuss its origins, and provide a "fix" for it. In Section 6 we describe the conjecture attempted in PVS and then our generation of a counterexample using the Symbolic Analysis Laboratory (SAL) [11]. Section 7 outlines the specific challenge and describes some metrics for success. Concluding remarks are in Section 8.

2 The ROBUS IC Protocol

We begin by describing the background of the family of protocols from which the ROBUS IC Protocol comes. Then after describing the architecture of the ROBUS, we describe the protocol's behavior itself.

Background 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 [12]. Lynch's textbook provides an introduction to these sort of protocols as well as pointers into the literature [13]. Many of these protocols have been formally verified, both by theorem proving [14–16] and by model checking [17].

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 redundancy. The architecture for the special case of three BIUs and three RMUs is shown in Figure 1. There must be a minimum of

[3] The associated files can be retrieved at ⟨`http://www.cs.indiana.edu/~lepike/pub_pages/disproving.html`⟩.

one BIU and one RMU, and there need not be an equal number of BIUs and RMUs.

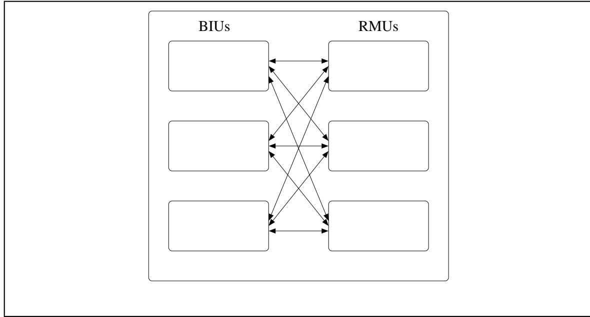

Fig. 1: The ROBUS Architecture

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* 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*. Every (non-faulty) node assigns every other node to exactly one class. We call the node being labeled the *defendant*. 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 [7]. 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

We 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 Section 4). In the following, a *benign message* is one that all nonfaulty nodes can detect came from a faulty node (see Section 3). The ROBUS IC Protocol is as follows (the message-passing events of the protocol are illustrated in Figure 2):

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 b 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 the result; otherwise, the majority value is the result.

3 Faults

Fault Classifications Faults result from innumerable occurrences including physical damage, electromagnetic interference, and "slightly-out-of-spec" communication [5]. 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 [18]. 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) or messages that arrive 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* [6], if it arbitrarily sends different messages to different receivers.

Fault Assumption A fault-tolerant protocol is designed to tolerate a certain number of faults of each 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 [19].

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 good BIUs b and good 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, let r_1 be a good RMU, 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*.*

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 r_1 declares n.*

These properties similarly hold for any two good RMUs with respect to a defendant n.

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 [7], 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.

Counterexample 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 Figure 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 [20]. 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

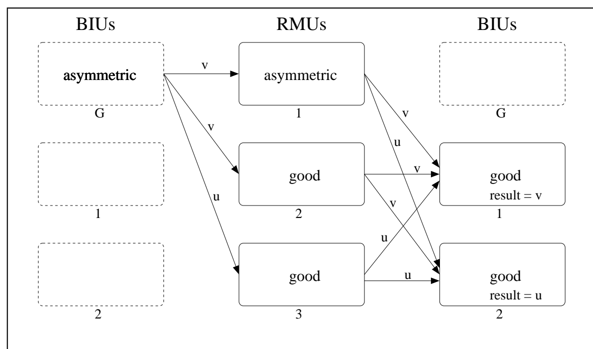

Fig. 2: An Instance of the ROBUS IC Protocol Violating Agreement

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 Section 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 by Miner et al. [21].

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 our approach, we use the PVS theorem proving system developed by SRI International [2]. We have used PVS to specify and verify other ROBUS protocols [21, 7]. The specification language of PVS is a strongly-typed higher-order logic, and 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.[4] A general discussion of the abstractions used in the model is provided elsewhere [22]. The following notation is used in the formal statements of the Agreement Conjecture and the unproved sequent in Figure 3 and Figure 4.

Variables and Parameters The parameters `B` and `R` are uninterpreted natural numbers. The set of BIUs and RMUs are indexed from 0 to `B - 1` and 0 to `R - 1`, and these sets of indicies are denoted `below(B)` and `below(R)`, respectively. Thus, the PVS specification is parameterized by the number of BIUs and RMUs, and a proof of correctness holds for any instantiation of these parameters that satisfy the hypotheses of the proof. Let `b1`, `b2`, `G` $\in$ `below(B)`, where `G` is used to designate the General. `F` is a variable over the set of *records* (i.e., named tuples [23]) that contain all of the diagnoses in the system. The ‘ operator provides record access. Thus, `F‘RB` denotes the collection of the BIUs' diagnoses against the RMUs, `F‘BB` denotes the BIUs' 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 diagnoses. `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. Finally, `msg` is an arbitrary message being broadcast by the General.

Functions and Relations The following functions and relations appear in the sequent.

- `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.
- `IC_DMFA` is a formal statement of the IC DMFA described in Section 3.
- `all_correct_accs?` is a predicate formally stating the Diagnostic Assumptions defined in Section 4.

[4] The PVS models are more abstract than needed to model this protocol since the many of the same theories are generalized to model other ROBUS protocols [21].

- `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.
- `robus_ic` is a higher-order function that functionally models the ROBUS IC Protocol, as described in Section 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 function is essentially the composition of two functions modeling the two rounds of message passing (recall that we are modeling the protocol in the synchronous domain, so the rounds of message passing is the granularity at which time is modeled).

It is the convention of PVS to denote skolem constants with a trailing "`!n`," where `n` is some integer.

The Sequent The conjecture to be proved is shown in Figure 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`.

```
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)
```

Fig. 3: The Agreement Conjecture in PVS

Every branch of the conjecture in Figure 3 is discharged except for the branch ending in the single sequent in Figure 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.

6.2 Model Checking the Sequent

We use the Symbolic Analysis Laboratory (SAL) [24, 11], also developed by SRI International, to model check the protocol against the undischarged sequent. SAL is a family of model checkers that includes symbolic, bounded, and

```
[-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)
```

Fig. 4: The Unproven PVS Sequent

explicit-state model checkers, among other tools. The SAL language includes constructs such as recursive function definition, synchronous and asynchronous composition operators, and quantifiers over finite types. 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 [17]. Our model differs slightly 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.

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

$$\bigwedge \mathcal{A} \Longrightarrow \bigvee \mathcal{C} . \qquad (1)$$

This formulation is used to express the sequent in SAL and appears in Figure 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.

```
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]));
```

Fig. 5: The SAL Formulation of the Undischarged Sequent

Two additional statements in the LTL formulation are artifacts of how the protocol is modeled in the model checker, both of which come from Rushby's original formulation. 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 Section 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 `robus_ic`.

Thus, the conjecture in Figure 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 Figure 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

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

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. Such a state matches the informal description of the counterexample in Section 5.

Using SAL's symbolic model checker on a system with one gigabyte of memory and an AMD Athlon 2000+ processor, a counterexample like the one described in Section 5 is 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 Section 5 allows the Agreement proof to be completed [21], and SAL verifies the formula in Figure 5 in a model using the same number of BIUs and RMUs as used to find the counterexample (the fix is included as commented code in the SAL model available on-line[3]).

6.3 Remarks on our Approach

In our approach, we manually model the protocol and the requirements both in PVS and SAL. This is simultaneously advantageous and disadvantageous. Having to model the protocol and requirements in distinct languages provides an additional guard against modeling errors in each language. In particular, we wish to guard against false negatives, which are particularly easy to generate in model checkers.

A disadvantage is the additional work required to model the protocol and requirements twice. This approach is not feasible to check numerous failed proof conjectures in a proof attempt.

Some limitations of this approach are inherent to the limitations of symbolic model checking, in general. A model checker is useful when the system can be modeled as a finite-state state machine, and the requirements to be proved can be modeled in a temporal logic. As well, a counterexample may exist, but be beyond the computational limits of the model checker and the computer on which it executes.

Despite these limitations, we believe this approach approximates the current state-of-the-art to verify and discover bugs for a protocol like the one described.

Related Work Much previous work that integrates model checking and interactive theorem proving has focused on using model checking to automate proving rather than on disproving [25, 26]. Some theorem provers have embedded model checkers (both PVS and ACL2 contain embedded μ-calculus model checkers [27, 28]). Most related to our work is a study in which resolution-based theorem proving and model checking are used to discover counterexamples to proof obligations [29]. Our work differs in that we present a reasonably intricate protocol whereas a small illustrative example is presented therein. As well, the focus therein is on *automated* theorem proving; our focus is on using model checking to facilitate *interactive* theorem proving.

7 The Challenge

The challenge is as follows:

> *From a parameterized specification of the protocol (from which a general proof can be obtained), provide a concrete instance of the bug in a way that requires as little effort from the user as possible.*

This section describes how our approach could be improved as well as speculation about other approaches.

Specification Languages In our specific case, the specification languages of PVS and SAL are similar, and it is a goal of SRI to develop translators between them [30]. Building a translator between the languages is not trivial as the languages overlap, but neither is a subset of the other. The lack of a translator required us to model the protocol twice.

More generally, efficient disproving requires better integration between proving tools (e.g., theorem provers) and disproving tools (e.g., model checkers) [31, 32] regardless of the theorem prover or model checker choice made. As noted in Section 6, some theorem-provers contain model-checkers, but it is not clear that these model checkers can handle this case study.

Parameter Interpretation Even if a translator existed, the parameterized PVS specification is not immediately amenable to finite-state model checking: the parameters must be interpreted. For a system with many parameters or a large specification, the interpretation is tedious if done manually. The tedium is compounded by the need to find parameters small enough to make model checking feasible yet large enough to expose the counterexample.

A quickcheck approach, like the one that appears in Isabelle [33], may be sufficient to demonstrate a counterexample. However, doing so requires a reformulation of the problem since our formulation in PVS uses non-executable constructs (e.g., quantifiers and Hilbert's choice operator). How this could be done for a parameterized specification of the protocol that is not implementation-specific is unknown to the authors.

Proof Calculus Translation A minor but relevant task in our approach is the translation of a failed conjecture in the sequent calculus into an LTL formula. We hope for this task to be automated, too (for whatever proof calculus and temporal logic is used).

Proving and Disproving Ideally, both proving and disproving would be automatic. Automated disproving appears to be an easier challenge to meet than proving. That said, recent advances in *satisfiability modulo theories* (SMT) provers [34] hold promise. Fault-tolerant protocols have generally been good candidates for mechanical theorem proving given their criticality and complexity, but SMT technology may provide a more automated approach. Currently, SMT provers are not capable of proving a fully-parameterized specification of the protocol presented, but recent applications of SMT to verifying fault-tolerant and real-time protocols are promising [35, 20]. For these sort of verifications, SMT is particularly useful when combined with bounded-model checking to do highly-automated induction proofs of safety properties over infinite-state systems [36]. We hope the SMT community also takes up the protocol presented as a challenge problem in parameterized proof as well as counterexample generation.

We do not believe that automated first-order theorem provers alone can prove the correctness of the SPDIER IC Protocol. The protocol's proof of correctness is parameterized in the number of BIUs and RMUs. Furthermore, the proof of correctness depends on reasoning about nontrivial mathematical facts (e.g., the IC DMFA). A first-order theorem prover could possibly be used to derive a counterexample for a fixed specification, replacing the model checker in our approach. Nonetheless, a similar interactive consistency protocol has been specified and verified in ACL2, an interactive first-order theorem prover [14]. Also, some initial work has been done to translate the PVS specification of the SPIDER IC Protocol into a form suitable for SAT solving [37].

Effort Requirements Given the criticality of the correct design of SPIDER and similar safety-critical embedded systems, their designers are willing to invest a great deal of effort to gain high assurance of their correctness. Therefore, the acceptable level of effort required to obtain counterexamples is relatively high, as evidenced by the use of interactive theorem proving in the first place. Certainly the upper bound on the acceptable level of effort to uncover a counterexample is the time it takes a user to diagnose the error in the failed proof attempt in the theorem prover. This effort varies according to experience with the theorem prover, expertise in the domain, the proof infrastructure, and the specific reason the proof has failed.

8 Conclusion

We have presented a subtly-flawed fault-tolerant protocol, its attempted verification by theorem proving, and our use of a model checker to demonstrate that a bug in the protocol prevents us from completing the verification. As noted, we believe our approach approximates the best possible with today's technology.

More generally, a variety of real-time fault-tolerant protocols have been designed for SPIDER and are described in detail [4]. Most of these protocols have been verified using theorem proving and model checking with SMT [38, 21, 20]. These protocols are all suitable case studies to demonstrate novel verification tools and techniques.

Acknowledgments

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 and Jeffrey Maddalon. We thank our anonymous reviewers of the DISPROVING Workshop for their helpful and detailed comments. As noted, the SAL model was adopted from work by Rushby [17].

References

1. Pike, L., Miner, P., Torres, W.: Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM–2004–213278, NASA Langley Research Center (2004) Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html`.
2. Owre, S., Rusby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. IEEE Transactions on Software Engineering **21** (1995) 107–125
3. NASA Formal Methods Group: SPIDER homepage. Website (2004) Available at `http://shemesh.larc.nasa.gov/fm/spider/`.
4. Torres-Pomales, W., Malekpour, M.R., Miner, P.: ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center (2005)
5. Driscoll, K., Hall, B., Sivencrona, H., Zumsteg, P.: Byzantine fault tolerance, from theory to reality. In Goos, G., Hartmanis, J., van Leeuwen, J., eds.: Computer Safety, Reliability, and Security. Lecture Notes in Computer Science, The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP, Springer-Verlag Heidelberg (2003) 235–248
6. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of of the ACM **27** (1980) 228–234
7. Geser, A., Miner, P.: A formal correctness proof of the SPIDER diagnosis protocol. Technical Report NASA/CP-2002-211736, NASA Langley Research Center, Hampton, Virginia (2002) Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLSs).
8. Rushby, J.: Bus architectures for safety-critical embedded systems. In Henzinger, T., Kirsch, C., eds.: EMSOFT 2001: Proceedings of the First Workshop on Embedded Software. Volume 2211 of Lecture Notes in Computer Science., Lake Tahoe, CA, Springer-Verlag (2001) 306–323
9. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. IEEE Computer **30** (1997) 75–82 Available at `citeseer.ist.psu.edu/hsueh97fault.html`.
10. Kopetz, H.: Real-Time Systems. Kluwer Academic Publishers (1997)
11. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: Computer-Aided Verification, CAV 2004. Volume 3114 of Lecture Notes in Computer Science., Boston, MA, Springer-Verlag (2004) 496–500
12. Davies, D., Wakerly, J.F.: Synchronization and matching in redundant systems. IEEE Transactions on Computers **27** (1978) 531–539
13. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
14. Young, W.D.: Comparing verification systems: Interactive consistency in ACL2. IEEE Transactions on Software Engineering **23** (1997) 214–223
15. Bevier, W., Young, W.: The proof of correctness of a fault-tolerant circuit design. In: Second IFIP Conference on Dependable Computing For Critical Applications. (1991) Available at `http://citeseer.ist.psu.edu/bevier91proof.html`.
16. Lincoln, P., Rushby, J.: Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In: Compass '94 (Proceedings of the Ninth Annual Conference on Computer Assurance), Gaithersburg, MD, IEEE Washington Section (1994) 107–120 Available at `http://www.csl.sri.com/papers/compass94/`.
17. Rushby, J.: SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Technical Report CSL Technical Note, SRI International (2004) Available at `http://www.csl.sri.com/users/rushby/abstracts/om1`.
18. Thambidurai, P., Park, Y.K.: Interactive consistency with multiple failure modes. In: 7th Reliable Distributed Systems Symposium. (1988) 93–100

19. Butler, R.W.: The SURE approach to reliability analysis. IEEE Transactions on Reliability **41** (1992) 210–218
20. Pike, L., Johnson, S.D.: The formal verification of a reintegration protocol. In: EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software, New York, NY, USA, ACM Press (2005) 286–289 Available at `http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html`.
21. Miner, P., Geser, A., Pike, L., Maddalon, J.: A unified fault-tolerance protocol. In Lakhnech, Y., Yovine, S., eds.: Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT). Volume 3253 of Lecture Notes in Computer Science., Springer (2004) 167–182 Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unified.html`.
22. Pike, L., Maddalon, J., Miner, P., Geser, A.: Abstractions for fault-tolerant distributed system verification. In Slind, K., Bunker, A., Gopalakrishnan, G., eds.: Theorem Proving in Higher Order Logics (TPHOLs). Volume 3223 of Lecture Notes in Computer Science., Springer (2004) 257–270 Available at `http://www.cs.indiana.edu/~lepike/pub_pages/abstractions.html`.
23. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. SRI International. Version 2.4 edn. (2001) Available at `http://pvs.csl.sri.com/manuals.html`.
24. SRI International: Symbolic analysis laboratory SAL (2004) Available at `http://sal.csl.sri.com/`.
25. Rushby, J.: Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In Dams, D., Gerth, R., Leue, S., Massink, M., eds.: Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops. Volume 1680 of Lecture Notes in Computer Science., Trento, Italy, and Toulouse, France, Springer-Verlag (1999) Available at `http://www.csl.sri.com/papers/spin99/`.
26. Havelund, K., Shankar, N.: Experiements in theorem proving and model checking for protocol verification. In: Proceedings of Formal Methods Europe FME'96. Lecture Notes in Computer Science, Springer (1996)
27. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Prover Guide. SRI International. Version 2.4 edn. (2001) Available at `http://pvs.csl.sri.com/manuals.html`.
28. Manolios, P.: Chapter 7: Mu-Calculus Model-Checking. In: Computer Aided Reasoning: ACL2 Case Studies. Self-Published (2002)
29. Bicarregui, J.C., Matthews, B.M.: Proof and refutation in formal software development. In: 3rd Irish Workshop on Formal Methods (IWFM'99). (1999)
30. SRI Computer Science Laboratory: Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, SRI International, Menlo Park, CA 94025 (2003)
31. Johnson, S.D.: View from the fringe of the fringe. In Margaria, T., Melham, T., eds.: 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Volume 2144 of Lecture Notes in Computer Science., Springer-Verlag (2001) 1–12
32. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N.: Integrating verification components. In: Verified Software: Theories, Tools, Experiments. (2005)
33. Nipkow, S.B.T.: Random testing in Isabelle/HOL. In Cuellar, J., Liu, Z., eds.: Software Engineering and Formal Methods (SEFM 2004), IEEE Computer Society (2004) 230–239
34. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In Etessami, K., Rajamani, S., eds.: 17th International Conference on Computer Aided Verification, Springer (2005) 20–23
35. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: Formal Techniques in Real-Time and Fault-Tolerant Systems. Volume 3253 of Lecture Notes in Computer Science., Grenoble, France, Springer-Verlag (2004) 199–214 Available at `http://fm.csl.sri.com/doc/abstracts/ftrtft04`.
36. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In Voronkov, A., ed.: Computer-Aided Verification, CAV 2003. Volume 2725 of Lecture Notes in Computer Science., Springer-Verlag (2003) 14–26
37. Sinz, C.: Propositional translation of PVS specifications. Talk slides (2004) Talk held at the National Institute of Aerospace/NASA Langley Research Center. Available at `http://www-sr.informatik.uni-tuebingen.de/~sinz/pdf/prop_trans.pdf`.
38. Pike, L.: Formal Verification of Time-Triggered Systems. PhD thesis, Indiana University (2006) Available at `http://www.cs.indiana.edu/~lepike/phd.html`.