# A Test Generation Framework for Distributed Fault-Tolerant Algorithms

Alwyn Goodloe
National Institute of Aerospace

Corina S. Păsăreanu
Carnegie Mellon University/NASA Ames

David Bushnell
TracLabs/NASA Ames

Paul Miner
NASA Langley

## ABSTRACT

Heavyweight formal methods such as theorem proving have been successfully applied to the analysis of safety critical fault-tolerant systems. Typically, the models and proofs performed during such analysis do not inform the testing process of actual implementations. We propose a framework for generating test vectors from specifications written in the Prototype Verification System (PVS). The methodology uses a translator to produce a Java prototype from a PVS specification. Symbolic (Java) PathFinder is then employed to generate a collection of test cases. A small example is employed to illustrate how the framework can be used in practice.

## 1. INTRODUCTION

Verification and validation of distributed fault-tolerant systems is a continuing challenge for safety-critical systems. In order to provide V&V support for distributed fault-tolerant algorithms, we are exploring a combination of technologies. Ultimately, fault tolerance consists of establishing and maintaining consensus between distributed computational resources, especially when a bounded subset of these resources is faulty. A full analysis requires an understanding of both the distribution and failure modes of the sensors, effectors, and computational resources. There are several different valid ways of architecting these systems to meet fault-tolerance requirements. This compounds the problem of providing a collection of tools supporting V&V activities. Substantiating fault-tolerance claims requires a combination of analysis and test. We are researching an approach to V&V where the test-vectors are generated from formal models expressed using SRI's Prototype Verification System (PVS).

Given that safety-critical systems are usually developed to exacting certification criteria, system failures are often the result of unanticipated events such as dirty voltage on a bus or a hardware fault. A formal model of a fault-tolerant system should explicitly model the faults that the system can handle and a testing regime should validate that the system does indeed process these as advertised. Thus testing the actual system must include injecting faults into the system. One preferred way to do this is to employ a test harness that can inject data into the system so that it appears as if a fault has occurred. Given a PVS specification, we are developing a methodology for generating these tests automatically.

Rather than develop a new tool suite from scratch, we apply two existing tools to the task of test case generation. A PVS to Java translator, developed to create executable prototypes from the specification, is applied to generate a realization of the protocol preserving the correctness properties shown to hold in the PVS model. Symbolic Java PathFinder, a model-based automated software test generation tool [9], is then used to generate test vectors that can be used by V&V engineers to test actual protocol implementations.

The paper is organized as follows. We first introduce a case study of a failure in the space shuttle. The next section provides an overview of fault-tolerance. This is followed by an overview of a PVS model of a small consensus protocol. Next is a brief description of the PVS-to-Java translator. We then discuss Symbolic Java PathFinder. Section 7 discusses test case generation for the protocol. Finally, we discuss related works and conclude.

## 2. FAILURE IN THE SPACE SHUTTLE

The Space Shuttle's data processing system has four general purpose computers (GPC) that operate in a redundant set. There are also twenty three multiplexer de-multiplexers (MDM) units aboard the orbiter, sixteen of which are directly connected to the GPCs via redundant shared busses. Each of these MDMs receive commands from guidance navigation and control (GNC) running on the GPC and acquires requested data from sensors attached to it, which is then sent to the GPCs. In addition to their role in multiplexing/demultiplexing data, these MDM units perform analog/digital conversion. Data transfered between the GPC and MDMs is sent in the form of serial digital data.

The GPCs execute redundancy management algorithms that

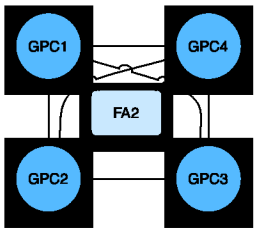


**Figure 1: Shuttle Data Processing System (GPCs and FA2)**

include a fault detection, isolation, and recovery (FDIR) function. During the launch of shuttle flight Space Transportation System 124 (STS-124), there was reportedly a pre-launch failure of the fault diagnosis software due to a "non-universal I/O error" in the second flight aft (FA2) MDM [3], which is polled by the GPCs as shown in Figure 1. According to [3, 4], the events unfolded as follows:

- A diode failed on the serial multiplexer interface adapter of the FA2 MDM.
- GPC 4 receives erroneous data from FA2. Each node votes and views GPC 4 as providing faulty data. Hence GPC 4 is voted out of the redundant set.
- Three seconds later, GPC 2 also receives erroneous data from FA2. In this case, GPC 2 is voted out of the redundant set.
- In accordance with the Space Shuttle flight rules [22], GPC 2 and GPC 4 are powered down.
- GPC 3 then reads FA2's built-in test equipment and determines that GPC 3 is faulty at which point it too is removed from redundancy set leaving only GPC 1 at which time engineers terminated the work and the problem with FA2 was isolated and the unit replaced.

The above set of events sequentially removed good GPC nodes, but failed to detect and act on the faulty MDM. Based on the analysis reported in [4], it appears the system had a single point of failure. Even though the nodes were connected to the MDM via a shared bus, conditions arose where different nodes obtained different values from MDM FA2.

## 3. FAULT-TOLERANCE

The terms 'failure', 'error', and 'fault' have technical meanings in the fault-tolerance literature. A *failure* occurs when a system is unable to provide its required functions. An *error* is "that part of the system state which is *liable to lead to subsequent failure*," while a *fault* is "the *adjudged or hypothesized cause* of an error" [20]. For example, a sensor may break due to a fault introduced by overheating. The sensor reading error may then lead to system failure.

We are primarily concerned with architectural-level fault-tolerance [8]. A *fault-tolerant system* is one that continues to provide its required functionality in the presence of faults. A fault-tolerant system must not contain a *single point of failure* such that if that single subsystem fails, the entire system fails (for the faults tolerated). Thus, fault-tolerant systems are often implemented as distributed collections of nodes such that a fault that affects one node or channel will not adversely affect the whole system's functionality.

Faults can be classified according to the hybrid fault model of Thambidurai and Park [28]. Here, we characterize the faults of a node in a distributed system based on the messages other nodes receive from it. The same characterization could be made of channels. First, a node that exhibits the absence of faults is *non-faulty* or *good*. A node is called *benign* or *manifest* if it sends only *benign messages*. Benign messages abstract various sorts of misbehavior that are reliably detected by the transmitter-to-receiver fault-detection mechanisms implemented in the system. For example, a message that suffers a few bit errors may be caught by a cyclic redundancy check. In synchronized systems, nodes that send messages received at unexpected times are also considered to be benign. A node is called *symmetric* if it sends every receiver the same message, but these messages may be arbitrary. A node is called *asymmetric* or *Byzantine* if it sends different messages to different receivers, and at least one of the messages received is not detectably faulty [19]. (Note that the other message may or may not be incorrect.)

We model faulty behavior exclusively within the communication model. This leads to an observational classification of fault effects. The fault effect classification model we employ is derived from the Azadmanesh and Kieckhafer [1] generalization of the Thambidurai and Park [28] hybrid fault model. These fault classifications are from the perspective of the receivers. Each source node is classified according to its worst observed error manifestation. The classification is a function of both the behavior of the node and how that behavior is perceived by a specified collection of observers.

The nodes are classified according to the following definitions:

**good** All receivers receive correct values.

**omissive symmetric** All receivers receive either correct values or manifestly incorrect values (including the possibility of no message at all). All receivers observe the same pattern of messages.

**omissive asymmetric** Some receivers may receive correct messages, while others may receive manifestly incorrect values.

**transmissive symmetric** All receivers observe the same pattern of messages. Messages may be incorrect.

**fully transmissive asymmetric (Byzantine)** Receivers may receive arbitrarily different values.

Some protocols are defined to operate correctly under the assumption that the possible fault behaviors are limited to specific subsets of these possible observable fault manifestations. Whenever this is done, there is an obligation to

validate the fault hypotheses, since the various theoretically possible fault manifestations have been observed in both laboratory and deployed systems [11]. Likewise, if a system purports to continue to operate correctly under some bounded number of Byzantine faults, this too must be validated.

In the case study given in Section 2 the nodes were connected to the MDM via a shared bus, yet conditions arose where different nodes obtained different values from MDM FA2. This is consistent with the MDM FA2 failing in a Byzantine fashion sending different values to the GPCs using the topology in Figure 1. Note that the triple-redundancy voting scheme employed in the case of the shuttle would have masked many faults, but not Byzantine faults. The designers may have simply assumed that such Byzantine faults were too unlikely to occur to warrant the additional complexity needed to handle them, yet they appear to have occurred in practice. This illustrates the need for the V&V process to test not only the fault model advertised by the system, but to test the assumptions built into that fault model.

V&V of fault-tolerant systems requires that the test engineer fully exercise the faults purportedly covered by the fault model. In cases where the system's fault model does not accommodate Byzantine faults, it may still be desirable from the V&V perspective to demonstrate that such assumptions actually hold and if not, how a system functions in the presence of such faults. The tests must include faults that are physically possible, but not logically anticipated. Such faults may arise from hardware failures, radiation faults, as well as from traditional inputs. So the V&V engineer must produce a range of inputs, a range of faults, and inject the said faults into the system and observe their effects.

# 4. FORMAL MODEL

Consider the situation of a *transmitter* node (this could be a sensor as in the shuttle example) that sends data to a number of *receiver* nodes. As we have seen, the transmitter, a receiver, or the interconnect may suffer a fault that causes a receiver to compute a received value that differs from those received by the other receivers. A simple variant of the oral-messages Byzantine protocol [19] can be employed to mask a bounded number of these errors. The protocol that we consider uses *Relay* nodes. The data flow of the protocol is illustrated in Figure 2, where the transmitter sends a message to each of the relay nodes, which, in turn, sends the value they received to each of the end nodes. At the receiver, a majority vote is performed on the values received. If the number of faults are appropriately bounded, then the nodes have achieved consensus.

We built a small PVS model of this protocol, where each node has a core functional component and network interface (NIC) (receiver/sender) components. This architecture is illustrated in Figure 3. The components are connected by FIFO queues. A message sent from the transmitter to the relay nodes is placed in the queue to the transmitter's NIC sender, which places the message in queues connecting it to the receiver NICs at each of the relay nodes. The relay module removes the message from the NIC receiver queue and places it into a queue leading to the NIC sender, which copies the message into the queue of each receiver.

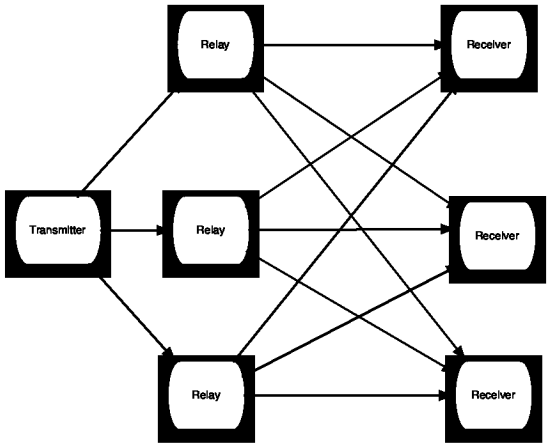


**Figure 2: Simple Oral Messages Protocol**

We assume a synchronous execution model with the nodes assumed to work in lock-step as if on a global clock. For instance, the transmitter sends its value to its NIC sender at clock tick 1, the value is sent to the relay node's NIC receiver at tick 2, etc.

In the fault model for our system, the transmitter is assumed to be subject to fully transmissive asymmetric errors. The relay nodes are assumed to be more reliable, say due to the use of redundant pairs, but subject to omissive asymmetric faults.

## 4.1 NIC Receiver

We now consider the PVS model of the NIC receiver in some detail. The NIC receiver is parameterized by the number of inbound communication channels, `maxsize`, and the global time at which the state machine is to execute. The messages are assumed to be of type `Frame`. The state of the NIC receiver is formed from the product of the following:

- `wires` : `below(maxsize)` $\rightarrow$ `fifo[Frame]`. The wires connect the receiver NIC to the sender NIC. As illustrated in Figure 3 each queue in the sequence is connected to a different node.
- `from_nic` : `below(maxsize)` $\rightarrow$ `fifo[Frame]`. This sequence of queues is used to pass the data received to the relay or end receiver state machines.
- `enabled` : `below(maxsize)` $\rightarrow$ `boolean`. If `enabled`($i$) is true, then the source node $i$ (the source node attached to `wires`($i$)) is assumed to be valid. The value can be false for a number of reasons including messages being dropped or garbled.
- `pc` is the current global clock value.

At this time, we do not model details of buffering, CRC checks etc. Instead, we focus on capturing the fault model at this node. The state machine defining this component has the signature

`NICReciver` $\times$ `AllReceiveActions` $\rightarrow$ `NICReceiver`,

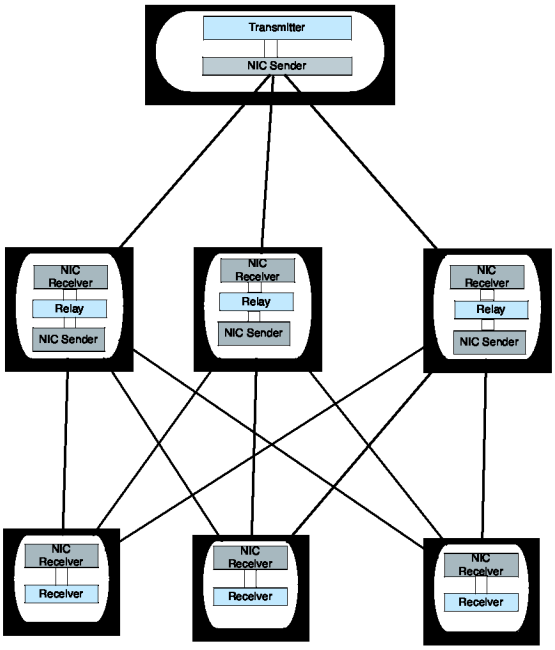


**Figure 3: Structure of the Model**

where `AllReceiveActions` is defined as

```
AllRecActions:TYPE = below(maxsize) → Actions.
```

The PVS datatype `Actions` defines the allowed faults and is defined as follows:

```
Actions : DATATYPE
  BEGIN
    Good             : Good?
    Garbled          : Garbled?
    Sym(frame:Frame) : Sym?
    Asym(frame:Frame): Asym?
  END Actions.
```

Recall that `wires` hold messages from each incoming channel and faults are not modeled anywhere else in our model so a message is not corrupted or lost in transit. Instead, we use `AllReceiveActions` to inject faults at the NIC receiver. This type acts as a filter so if `AllRecActions`($i$) has the value `Good`, the message received is assumed to be good and is relayed on, the value `Garbled` is treated as a benign fault or a dropped message and would be dropped, the value `Sym(frame)` would result in using the value `frame`, with the same value sent to each node, and `Asym(frame)` would result in the value `frame` being placed in the `fom_nic`($i$) queue, where each $i$ may get a distinctly different value.

Each of the fault model classifications has an associated PVS type that constrains the values of the filter. For instance good messages are defined by the type:

```
AllGoodRecAction: TYPE =
{f:AllRecAction| ∀ (i:below(maxsize)): Good?(f(i))},
```

omissive asymmetric faults are defined by the type:

```
OmissAsymRecAction : TYPE = {f : AllRecActions |
 ∀ (i:below(maxsize)) : Good?(f(i)) ∨ Garbled?(f(i))}
```

and transmissive asymmetric faults are defined by the type:

```
TransAsymRecAction: TYPE =
{f: AllRecActions | ∀ (i:below(maxsize)):Asym?(f(i))}.
```

The PVS code for updating the `from_nic` component of the `NICReceiver` state is given as follows.

```
λ(i:below(maxsize)):
 CASES a(i) OF
   Good : IF  ¬empty_fifo?(s`wires(i)) THEN
          enqueue(topof(s`wires(i)),s`from_nic(i))
          ELSE empty_fifo
        ENDIF,
  Garbled : empty_fifo,
  Asym(frame) :  enqueue(frame,s`from_nic(i)),
  Sym(frame)  :  enqueue(frame,s`from_nic(i))
 ENDCASES,
```

which returns a new sequence of queues with the values determined by entries in the action sequence `a`. The state machine also removes the value received from each of the fifo queues comprising `wires`. If message $i$ was garbled, then the corresponding entry in `enabled` is set to false.

Above the state machines is a relational model where the components get connected together and that drives the state machine transitions. In the case of the NIC receiver, the fault model applied to a particular node is chosen. Existential quantification is used to model nondeterministic choice. Ideally we will want the faults to be created in a separate module from the system under test in order to model a test harness.

## 5. PVS TO JAVA TRANSLATOR

A PVS-to-Java translator [15] has been constructed as part of a collaborative effort between NIA and the Radboud University Nijmegen. The input to our code generator is a declarative specification written in PVS. Since we aim at a wide range of applications, we do not fix the target language. Indeed, the tool first generates code in Why, an intermediary language for program verification [12]. Our current prototype generates Java annotated code from the Why code. In the future, we may implement outputs for other functional and imperative programming languages.

In addition to enabling multi-target generation of code, another benefit of an intermediate language is that transformations and analysis that are independent from the target language can be applied to the intermediate code directly.

Consider the PVS datatype `Actions` defined in the Section 4. Each of the different actions becomes a subclass that extends the class `Actions` and has a constructor that takes the generic class `Frame` as an argument.

```
public class ReceiveAction <Data> {
 public class Actions { public Actions() {}}
 public class Sym extends Actions {
  FrameTh<Data>.Frame frame;
```

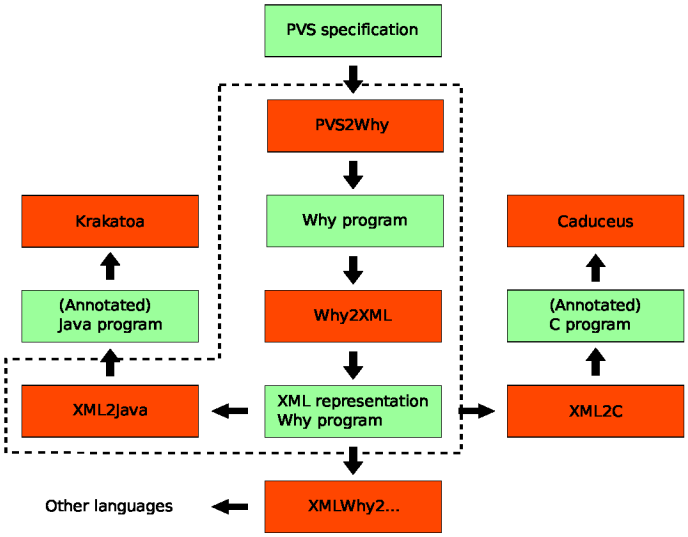


**Figure 4: Multi-target generation of verifiable code**

```
public Sym(FrameTh<Data>.Frame frame) {
 this.frame = frame; } }
```

The higher order use of defined functions is facilitated by a special **Lambda** class. This generic abstract class demands that an application function is supplied for each instance.

```
public abstract class Lambda<T1,T2> {
  abstract public T2 apply(T1 obj); }
```

For all defined functions in PVS, a higher order version is generated that satisfies the requirements of the **Lambda** class.

```
public Lambda<Actions,Boolean> SymRecognizer =
 new Lambda<Actions,Boolean>(){
  public Boolean apply(final Actions actions){
   return SymRecognizer(actions);}};}
```

All functions are translated into currified syntax. This way it is possible to translate all higher order uses of functions, including partial application, into working Java programs.

In its current phase of development, the translator can only translate PVS specifications that are written in functional style, in particular, as a state machine. Relational specifications, which typically model nondeterministic behavior, must be hand coded. The existential quantification in the specification, which is used to nondeterministically generate actions, gets replaced by hooks into the Symbolic Java PathFinder tool.

## 6. SYMBOLIC (JAVA) PATHFINDER

For test case generation, we will use Symbolic (Java) PathFinder (SPF) [9], a symbolic execution framework built on top of the Java PathFinder model checker [23]. SPF combines symbolic execution and constraint solving techniques for the automated generation of test cases that achieve high coverage. Symbolic PathFinder implements a symbolic execution framework for Java byte-code. It can handle mixed integer and real inputs, input data structures and strings, as well as multi-threading and input pre-conditions.

Symbolic execution [18] is a well-known program analysis technique that uses symbolic values instead of actual data as inputs and symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition ($PC$), and a program counter. The path condition is a boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. These conditions can be solved (using off-the-shelf constraint solvers) to generate test cases (test input and expected output pairs) guaranteed to exercise the analyzed code.

Symbolic PathFinder implements a non-standard interpretor for byte-codes on top of JPF. The symbolic information is stored in attributes associated with the program data and it is propagated on demand during symbolic execution. The analysis engine of JPF is used to systematically generate and explore the symbolic execution tree of the program. JPF is also used to systematically analyze thread interleavings and any other forms of non-determinism that might be present in the code; furthermore JPF is used to check properties of the code during symbolic execution. Off-the-shelf constraint solvers/decision procedures are used to solve mixed integer and real constraints. We handle loops by putting a bound on the model-checker search depth and/or on the number of constraints in the path conditions.

By default, Symbolic PathFinder generates vectors of test cases, each test case representing input-output vector pairs. In order to test reactive systems, such as the fault tolerance protocols that we are studying here, we have extended Symbolic PathFinder to also generate test sequences (i.e., sequences of test vectors) that are guaranteed to cover states or transitions in the models (other coverages such as condition, or user-defined are also possible). This works by instructing Symbolic PathFinder to generate and explore all the possible test sequences up to some user pre-specified depth (or until the desired coverage is achieved) and to use symbolic, rather than concrete, values for the input parameters.

## 7. PRELIMINARY RESULTS: TEST CASE GENERATION

We have begun applying Symbolic PathFinder to the Java code generated from the PVS specifications for the case study described in Section 4. Each of the PVS models for the components of the system (wires, senders, receivers, relays, and so on) was translated into a Java class through the process described in Section 5. We applied SPF to these classes combined with the hand-written driver code (also described in Section 5).

Both the connections among the components and the model's execution policy are implemented in the driver code. The Java code that was automatically derived from the PVS specifications does not assume any particular inter-component connections or execution policy. Since this case study initially assumes a synchronous execution policy with a global clock, our first implementation of a driver is a simple single-threaded model which can be outlined as:

```
... wire components together ...

for (int pc=0; pc<maxPc; pc++) {
  sender1.step();
  nicSender1.step();
  nicReceiver1.step();
  ...    }
```

To perform test case generation, we identified the *inputs* to the protocols as being the different types of faults that can be injected on the receiver side to test the fault tolerance behavior. In the PVS model, the four possible component fault types (**Good**, **Garbled**, **Symmetric**, **Asymmetric**) are introduced into the NIC Receiver through the **Actions** PVS datatype (see Section 4.1). However, our Java code simplifies this by implementing the four fault types as simple integers in the range **0,..,3**. This is only for convenience — it does not materially affect the model.

As a second simplification for this initial trial, we modeled only a single sender wired to two receivers. Future runs will expand the model to cover the full case study shown in Figure 3.

In order to generate test cases with SPF, we needed to annotate the Java code so that SPF knows that the integer **Actions** are the symbolic variables for which path conditions must be derived. This is easily done. The original driver code which sets up the array of actions (i.e. possible faults):

```
int[] actions = new int[maxsize];
for (int i = 0; i < maxsize; i++)
  actions[i] = ...some fault type in 0,1,2,3...;
```

is modified to tell SPF that the integers in **actions[]** are to be treated symbolically:

```
int[] actions = new int[maxsize];
for (int i = 0; i < maxsize; i++)
  String symVarName = "rcvr-" + (rcvrID++) "-error";
  actions[i] = Debug.getSymbolicInt(0,3,symVarName);
```

**Debug.getSymbolicInt (min, max, name)** is a utility method in SPF which tells the model checker to generate a symbolic integer named **name** whose range is **min...max**. This is the only annotation needed to apply SPF.

Running SPF then produces the test inputs from the path conditions:

```
Constraint 1:
rcvr-1-error == Good &&
rcvr-0-error == Good

Constraint 2:
rcvr-1-error == Garbled &&
rcvr-0-error == Good

...

Constraint 15:
rcvr-1-error == Symmetric &&
rcvr-0-error == Asymmetric

Constraint 16:
rcvr-1-error == Asymmetric &&
rcvr-0-error == Asymmetric
```

We then ran the Java code with these sixteen tests and measured the coverage. In some Java classes the coverage was less than 100%. Examining the code that was not executed showed that most of statements were in code that is not needed for the simple example used for this initial trial. A few unexecuted lines are significant and are the result of problems in the handwritten driver code. These problems will be addressed in future our future work.

# 8. RELATED WORK

Almost all of the major theorem provers provide some form of code generation their specification language. For instance, theorem prover Isabelle/HOL even provides two code generators [2, 16], ACL2's [17] specification language *is* a subset of Common Lisp, and Coq [5] has a generator [21] that extracts lambda terms and translates them in either Haskell or OCaml. Unlike the generator we use in this paper, these languages are all functional programming languages.

While there has been a lot of work on specification-based testing and test case generation [10, 13, 14], there has been little work focusing on bridging the gap between theorem proving and testing. The HOL-TestGen system [7] generates unit tests from Isabelle [24] specifications. The literature currently focuses on generating tests for common libraries. Sewell *et al.* have constructed a tool that uses HOL specifications as an oracle for testing protocols [6], but their focus is not on test case generation. An experiment in using PVS strategies to create random test cases directly from PVS specifications is reported in [25].

The work presented here is also related to the use of formal methods (including theorem proving and model checking) for analyzing fault tolerance of circuits and systems [26, 27]. In contrast to these works, our goal is to leverage the effort of building and formally verifying models of such systems into testing actual implementations.

# 9. CONCLUSION AND FUTURE WORK

Formal methods are increasingly accepted in the fault-tolerant systems community as a means to analyze the correctness of a design under the assumption of a well specified fault model. Yet testing must be employed to validate an executable against a model. Furthermore, testing should explore whether assumptions built into the fault model do indeed hold. We present a methodology whereby a PVS formal model drives the creation of a Java executable prototype that can be used as reference implementation. Test cases can then be generated by applying static analysis techniques to the prototype implementation. The PVS-to-Java translator and the Symbolic PathFinder tools enable this process by automating much of the task.

The work reported in this paper is still preliminary and much work remains to be done. The PVS-to-Java translator is still evolving as features are added that will allow us to automatically translate more of our model. SPF is similarly evolving.

For instance, the ability to generate tests from the abstract datatype `Actions` is under development.

The results reported in Section 7 reflect experiments intended to test the feasibility of the methodology and to drive what features need to be added to our tools. Our next milestone is to be able to exercise the full protocol seen in Section 3. Furthermore, we need to extend the generation of test cases to also include the expected output (e.g. various observable protocol states) as our goal is to use these test cases to test actual implementations. The protocol under consideration is very basic. Once we have mastered the process for this example, we expect to focus on more sophisticated protocols such as fault-tolerant distributed clock synchronization.

## Acknowledgments

We thank Eric Cooper, Mike Lowry and César Muñoz for their comments. This work was partially supported by NASA Cooperative Agreement NCC1-02043.

## 10. REFERENCES

[1] M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

[2] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *LNCS*, pages 24–40. Springer, 2002.

[3] C. Bergin. Faulty MDM removed. NASA Spaceflight.com, May 18 2008. Available at `http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues` -faulty-mdm-removed/. (Downloaded Nov 28, 2008).

[4] C. Bergin. STS-126: Super smooth endeavor easing through the countdown to l-1. NASA Spaceflight.com, November 13 2008. Available at `http://www.nasaspaceflight.com/2008/11/sts-126-endeavour-easing-through-countdown/`. (Downloaded Feb 3,2009).

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[6] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review*, pages 265–276, Aug. 2005.

[7] A. Brucker and B. Wolff. Test-sequence generation with hol-testgen - with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *Tests and Proofs*, Lecture Notes in Computer Science 4454. Springer-Verlag, 2007.

[8] R. W. Butler. A primer on architectural level fault tolerance. Technical Report NASA/TM-2008-215108, NASA Langley Research Center, 2008.

[9] C. Pasareanu and P. Mehlitz and D. Bushnell and K. Gundy-Burlet and M. Lowry and S. Person and M. Pape. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *International Symposium on Software Testing and Analysis*, pages 15–26. ACM Press, 2008.

[10] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings FSE'99*, pages 285–302, Sept. 1999.

[11] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP*, Lecture Notes in Computer Science, pages 235–248. Springer, September 2003.

[12] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, UniversitÃl' Paris Sud, Mar. 2003.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineer ing*, pages 146–162. Springer-Verlag, 1999.

[14] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[15] A. E. Goodloe, L. Lensink, and C. Muñoz. From verified specifications to verifiable software. Technical report, National Institute of Aerospace, 2008.

[16] F. Haftmann and T. A code generator framework for Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07, 08 2007.

[17] M. Kaufmann, J. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. Available at `http://citeseer.nj.nec.com/lamport82byzantine.html`.

[20] J.-C. Laprie. Dependability—its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictability Dependable Computing Systems*, ESPRIT Basic Research Series, pages 3–24. Springer, 1995.

[21] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[22] NASA - Johnson Flight Center. Space shuttle operational flight rules Volume A, A7-104, June 2002. Available from `http://www.jsc.nasa.gov` (Downloaded Nov 28, 2008).

[23] NASA Ames. Java PathFinder Version 3.1.1 User

Guide.

[24] T. Nipkow, L. Paulson, and W. Wenzel. *Isabelle HOL - A proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science 2283. Springer Verlag, 2002.

[25] S. Owre. Random testing in pvs. In *Workshop on Automated Formal Methods*, 2006.

[26] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Trans. Softw. Eng.*, 25(5):651–660, 1999.

[27] S. A. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *Proc. Design Automation and Test in Europe (DATE)*, April 2007.

[28] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.