NASA/CR–2012-217764

Model-Driven Test Generation of Distributed Systems

Arvind Easwaran, Brendan Hall, and Kevin Schweiker
Honeywell International, Inc., Golden Valley, Minnesota

September 2012

NASA/CR–2012-217764

Model-Driven Test Generation of Distributed Systems

Arvind Easwaran, Brendan Hall, and Kevin Schweiker
Honeywell International, Inc., Golden Valley, Minnesota

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

September 2012

Acknowledgments

We would like to extend our thanks to Dr. Wilfried Steiner and Dr. Bruno Dutertre for providing support throughout the execution of this work.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

This report describes a novel test generation technique for distributed systems. Formal models and formal verification tools, specifically the Symbolic Analysis Laboratory (SAL) tool-suite from SRI International (SRI), a nonprofit research institute, are used to generate concurrent test vectors for distributed systems. These are initially within an informal test validation context and later extended to achieve full modified condition decision coverage (MC/DC) of the TTEthernet protocol operating within a system-centric context.

Contents

1 Introduction

This document details work performed under NASA Task Order NNL10AB32T, Validation and Verification of Safety-Critical Integrated Distributed Systems — Area 2.

This document is intended to satisfy the requirements for deliverable 5.1.10 under Task 4.1.2.1. It accompanies and documents Deliverable 5.1.11, which is a set of system models in electronic form (e.g., sal-atg files and the corresponding trace outputs).

1.1 Scope

This report documents a study that investigates the feasibility of generating system level test vectors for a distributed system, from a formal model of the system. The study is based on the TTEthernet communication protocol [1], specifically the TTEthernet start-up and integration protocol. This document assumes that the reader is familiar with the TTEthernet protocol. The reader is advised to read the TTEthernet protocol standard [1] before reading this document. This document does not re-iterate the content of the standard.

1.2 Background and Motivation

During TTEthernet protocol development, the development team made extensive use of model checking technology using the Symbolic Analysis Laboratory (SAL) tool suite from SRI. Before implementing any protocol code, formal models of the protocol core TTEthernet startup and synchronization services were implemented and validated using the SAL tool-suite. The in-line application of formal methods within the development cycle introduced significant benefits to the TTEthernet program.

Formalizing the protocol definition in a form suitable for capture within the SAL tool-chain made it possible to detect and address behavioral ambiguities early in the life cycle. The associated formalization of the protocol fault model also fostered significant dialog between the protocol and system design teams. This lead to early agreement and clarification of the expected protocol functionality and its associated services.

Validating the protocol using model checking technology resulted in the detection and removal of erroneous edge-case scenarios. Early modeling of the protocol, combined with the feedback in the form of simulation and model checking counter example traces, also enabled the design team to develop an early *working intuition* with respect to the complex emergent interactions of the distributed protocol behavior. This intuition proved invaluable when the first protocol hardware implementations were debugged in the development laboratory.

Although the TTEthernet formal models were greatly beneficial during early protocol definition, they were not used as part of a formal program verification. The models were utilized by the TTEthernet risk mitigation program, where limited fault-injection testing was performed to validate the behavior of early protocol prototype hardware against the protocol model. Formal equivalence testing between the hardware and the SAL model was not formally performed.

The formal testing and verification of TTEthernet consisted of two parts that treated the network switch and end-system as separate entities. Each component was individually verified using directed requirements driven test campaigns together with System Verilog-constrained random based testing. Neither of these formal verification activities targeted the integrated system behavior of all the TTEthernet components. [1]

[1] This was was largely due to the overhead associated with the verification environment that limited the simulation of large system configurations with multiple end-system and switching components

To explore the integrated system behavior, a network integration lab (NIL) was developed. The NIL included a large network test bed of 25+ end-systems and 17+ switches instrumented for fault injection. The emphasis within the NIL environment was system integration and the validation of higher level system properties. Therefore, the NIL based testing did not specifically target protocol branch coverage. The complexity of the TTEthernet protocol also complicated the NIL testing, as personnel faced a steep learning curve before they could generate or parse system test scenarios and results.

From this experience of developing TTEthernet, it became apparent that greater value may be gained from the initial formal modeling effort. If the formal model could be used as a reference for expected system behavior, it may assist the training and debugging activities of the NIL personnel. In addition, if the formal model could be used to generate integrated system level test scenarios, it may mitigate some of the risks associated with the separation of switch and end-system test campaigns that may miss subtle interactions among the distributed components.

Finally if the system level test generation scenarios could be restricted to remain consistent with the underlying fault hypothesis of the core protocol, the execution of the system test campaign may also be used to validate the protocol soundness. For example by presenting evidence that all of the protocol logic is required and that extraneous logic is not present. This restriction prevents artificial test scenarios from being used to justify coverage.

In the ideal scenario, an integrated system level test suite could be used to test the system-level interaction of the core protocol logic using expected real-world system level scenarios. Ideally such a test campaign could then be used to validate that the implemented protocol behavior matches the assumed behavior of the formal model for each protocol decision point. For this reason, the generation of formal protocol coverage tests using the model and comparison of the model-predicted behavior with the implemented behavior was a highly desirable goal.

To achieve the model and hardware comparison, it is necessary to bridge the gap between the abstract formal model and the lower level behavior of prototype hardware. In addition, since SAL expertise is not common within product development groups, it was desired to support the capability to quickly generate example scenarios from the model with minimal knowledge of SAL syntax and tooling.

1.3 Tools

The research team reviewed the capabilities of other test generation tools to investigate whether current tooling could meet the above criteria. In particular, the team reviewed the capabilities of the Honeywell Integrated Lifecycle Tools & Environment (HiLiTE)tool-chain [2]. HiLiTE is a mature and quantifiable tool that produces requirements-driven test vectors from Simulink and/or Stateflow models, nominally achieving in excess of 90% modified condition decision coverage (MC/DC) structural coverage. Although the HiLiTE tool has proven to be highly capable, accommodating very large and complex models, our review found that it it could not accommodate state concurrency, and therefore could not accommodate the distributed state of the TTEthernet components.

Therefore the sal-atg (SAL Automated Test Generation) tool chain from SRI was selected to perform the test generation study. The models presented in this report have been developed using the sal-atg tool. The version of the tool chain used was a pre-release candidate of SAL-3.1. It is available for download at: http://sal.csl.sri.com.

The SAL-3.1 tool chain introduces a new capability which uses the Lingeling SAT-solver as a backend to SAL. In our work,we found Lingeling presented significant performance improvements over the Yices solver that is packaged with SAL. The majority of the work described in this report uses the Lingeling tool chain that is available here: http://fmv.jku.at/lingeling.

All tools were hosted on a 64-bit OpenSuse Linux System http://www.opensuse.org/en/. The hardware system was configured with 12 Gigabytes of system RAM and at least 3 processing cores.

1.4 Report Overview and Structure

This report is organized into five sections, including this introduction. Section 2 presents the SAL model for the TTEthernet startup protocol, extended with priorities. Section 3 discusses some early proof and test scenario explorations that were performed on the extended model. Section 4 presents the instrumentations used on the extended model to achieve MC/DC coverage and discusses test findings. Section 5 presents our conclusions to date.

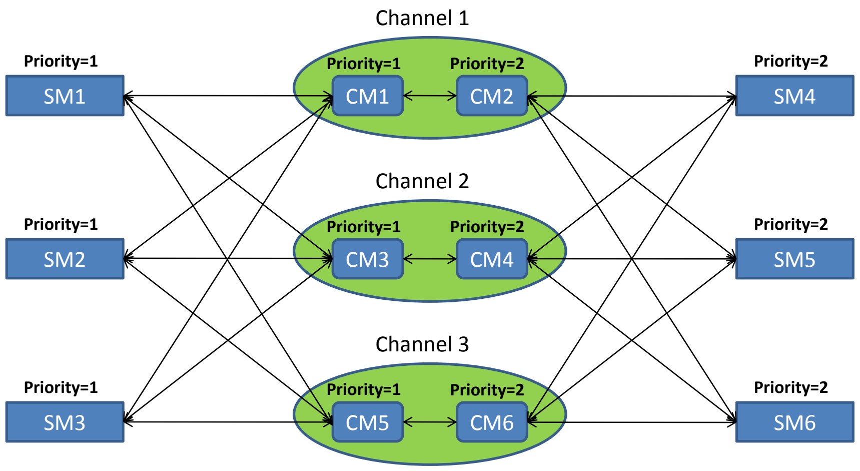

Figure 1. Modeled TTEthernet system

2 TTEthernet Startup Model and Model Extensions

In TTEthernet, the physical components on the network can be classified into three types depending on their functionality: 1) synchronization clients are end-systems that use the synchronized time-base to exchange information with other end-systems; 2) synchronization masters (SMs)are end-systems that are responsible for maintaining a synchronized time-base between components (in addition to exchanging information with other end-systems); and 3) compression masters (CMs) are switches on the network that are responsible for collecting, combining/compressing, and relaying synchronization messages between SMs in addition to relaying information between the end-systems.

Our work in modeling the behavior of the TTEthernet startup protocol was derived from the initial TTEthernet formal modeling performed by Wilfried Steiner as part of the Complexity Management for Mixed-Criticality Systems (CoMMiCS) research fellowship at SRI. For background context to the work presented in this section, refer to the CoMMics Deliverable [3].

The modeled system comprises six SMs and six CMs, arranged as shown in Figure 1. As part of the model extension we implemented some changes to improve model scalability. One such change was the modification of the `mem2nat` function. In the original model this was implemented using an `EXISTS` clause in `SAL`. We replaced this implementation with a recursive definition. As a result, we were able to scale the model from seven to twelve components. The model was extended to incorporate new states, so end systems can power on and sleep and CMs can power on late. Finally, the model was also extended to incorporate the TTEthernet-prioritized clock synchronization mechanism. In the example system shown in Figure 1, a *channel* is made up of a pair of CMs having different priorities. `SM1`, `SM2`, and `SM3` have a priority of one, and are directly connected to each of the CMs with priority 1 within the channels. Likewise, `SM4`, `SM5`, and `SM6` have a priority of 2 and are directly connected to each of the CMs with Priority 2.

2.1 SAL Model Detailed Description

The model is made up of six modules:

- The `synchronization_master` and `compression_master` modules model the protocol state machine for SM and CM, respectively. These state machines are described in detail in [1], and the initial SAL-based formal specification for these state machines is detailed in [3].
- The `priority_filter` module models the priority filtering mechanism for SMs, identifying at each step the set of messages to be delivered to an SM based on its priority. The SM priority mechanism and corresponding modeling constructs are described in detail in Section 2.2.
- The `sm_cm_connections` module models the connections between SMs and CMs, determining at each step the messages to be delivered. Fault modeling for the TTEthernet system is also implemented in this module.
- The `test_generator` module behaves like an observer of the system. It has complete knowledge of the state of the system at any point in time. This module is used to guide the system to specific states or to specify constraints on the behavior of the system.
- The `cluster` module integrates all these modules to create the system; it creates six instances of `synchronization_master`, `compression_master`, and `priority_filter` modules, and composes all the modules using the `synchronous` operator in SAL.

These modules are composed synchronously rather than asynchronously because of scalability issues. We believe that the model is correct from the perspective of our test generation goals, since the synchronous composition does not introduce any extraneous behavior in the protocol state machine.

SM and CM power on: In the extended model, the `test_generator` module can force any SM to be powered/re-powered or into a sleep state at any time. When powered/re-powered, the SM resets its internal state so that all the internal variables are re-initialized. The `test_generator` module can power-on any CM at any time, but it is prevented from forcing CMs into a sleep state. These power-on capabilities were sufficient to exercise our test generation goals.

Fault modeling: The extended model considers two faulty channels that exhibit inconsistent omissions — `channel 1` and `channel 2`. Each CM in a faulty channel can choose to not communicate with any subset of the SMs directly connected to it. The subset can be chosen non-deterministically. Note that if one of the CMs in the faulty channel breaks its connections with the SMs, then the other CM also cannot communicate with those SMs. A CM can independently choose to break either the outgoing or the incoming connection with any of the SMs directly connected to it. The two CMs within any channel are always assumed to be connected with each other. [2]

The SAL code for the outgoing connectivity between CMs is shown below. Identical code is used for the incoming connectivity between CMs. CMs within a channel are always connected to each other (*e.g.*, `CM1` and `CM2`), whereas CMs in different channels are never directly connected to each other.

[2]For the SM and CM coverage test scenarios in which there are no priorities in the model, each channel logically comprises of only one CM. The second CM is short-circuited so that there are direct connections between the first CM and all the end-systems in the model.

```
before_clique_connectivity_CM_out_to_CM IN
{a: ARRAY TYPE_CM_ids OF ARRAY TYPE_CM_ids OF BOOLEAN |
  a= [[i:TYPE_CM_ids] [[j:TYPE_CM_ids]
            ((i=1 OR i=2) AND (j=1 OR j=2)) OR
            ((i=3 OR i=4) AND (j=3 OR j=4)) OR
            ((i=5 OR i=6) AND (j=5 OR j=6))]]
};
```

The SAL code for the incoming connectivity between CMs and SMs is as shown below. Since Channels 1 and 2 are faulty, the connection from `CM1` and `CM3` to `SM1`, `SM2`, and `SM3` and the connection from `CM2` and `CM4` to `SM4`, `SM5`, and `SM6` are left unspecified. Since Channel 3 is not faulty, the connection from `CM5` to `SM1`, `SM2`, and `SM3`, and the connection from `CM6` to `SM4`, `SM5`, and `SM6` are forced to be valid. The rest of the connections are forced to be invalid, since they do not exist in our system.

```
before_clique_connectivity_CM_in_from_SM IN
{a: ARRAY TYPE_CM_ids OF ARRAY TYPE_SM_ids OF BOOLEAN |
  FORALL (i: TYPE_CM_ids, j: TYPE_SM_ids):
    ((i=1 OR i=3) AND (j=1 OR j=2 OR j=3)) OR
    ((i=5) AND (j=1 OR j=2 OR j=3) AND a[i][j]) OR
           ((i=2 OR i=4) AND (j=4 OR j=5 OR j=6)) OR
           ((i=6) AND (j=4 OR j=5 OR j=6) AND a[i][j]) OR
    ((i=1 OR i=3 OR i=5) AND (j=4 OR j=5 OR j=6) AND NOT a[i][j]) OR
    ((i=2 OR i=4 OR i=6) AND (j=1 OR j=2 OR j=3) AND NOT a[i][j])
};
```

In these models, the dependence between connectivity and CM or SM state has been abstracted. That is, the connectivity between a CM and an SM depends only on the existence of a path between those two components, possibly through other components, but does not depend on whether any of the components in the path are switched off. We can use the same connectivity module for both the priority scenarios—one has only one priority level and `CM2`, `CM4`, and `CM6` are switched off; the other has two priority levels and all CMs are powered on. For better understanding of our fault assumptions and connectivity between CMs and SMs, consider the connectivity scenario shown in Figure 2. As shown, `CM2`, `CM4`, and `CM6` are switched off, `CM1` can communicate only with `SM2`, and `CM3` can communicate only with `SM3`, while `CM5` can communicate with every SM; even though `CM6` is switched off.

In these models both the *high integrity* case and the *standard integrity* case are presented.

2.2 Adding Priorities to the Startup Model

This section describes how priorities were modeled in SMs and CMs using SAL and discusses how the behavior of the prioritized TTEthernet model accurately represents the hardware.

CM use of priorities in TTEthernet is straightforward; each CM has a pre-defined priority and responds only to synchronization messages from SMs that are sent with that priority. The CM ignores all other synchronization messages.

Each SM has a priority filter that sits between the end system and the incoming connections. The filter is configured to power on with a pre-defined *default priority* value. All synchronization messages sent by the SM use this default value for their priority. The filter can receive incoming messages at a priority level that is higher than or equal to the *current priority* of the SM. This

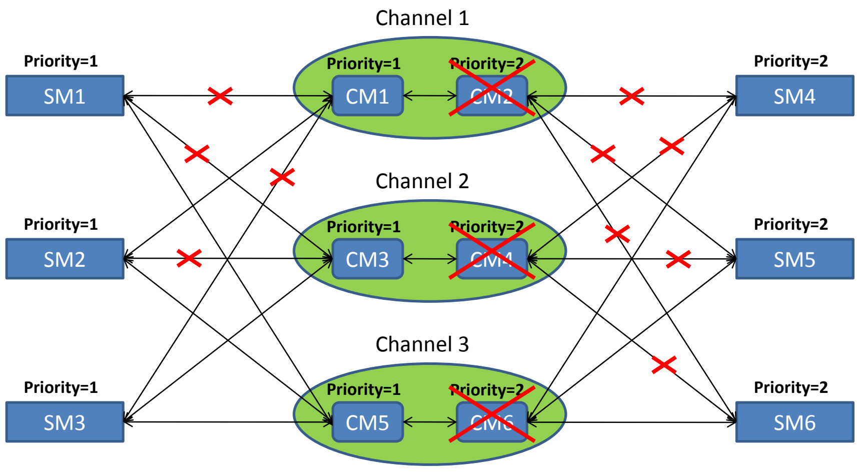

Figure 2. TTEthernet system connectivity

current priority is initialized to the default value at power on. Once an incoming message is accepted by the filter, the *current priority* of the SM is updated to be equal to that of the accepted message. The *current priority* is reset to the *lowest priority* value in the system if no incoming synchronization message passes the filter for a pre-defined timeout interval, denoted as `priority_-fallback_cycles` in the TTEthernet Specification [1]. Note that it is possible for the *current priority* value of an SM to be lower than its default priority value at certain time instants.

The protocol specification suggests that the priority filter may be configured in either autonomous or host-interactive mode. In *autonomous* mode, the filter automatically boosts the end-system priority by accepting higher priority synchronization messages as described above. In *host-interactive* mode, the filter boosts the priority and accepts higher priority messages only after the end-system acknowledges its request for the same. In the models presented here, only the autonomous mode of operation for the priority filter has been considered.

One of the challenges in modeling a TTEthernet prioritization scheme using SAL was to develop an abstract model of the protocol, while at the same time ensuring that the model accurately represents the behavior of the hardware. One abstraction technique that we adopted is the use of *transparent messages* being sent from CMs to SMs. Each transparent message abstracts a sequentialized, almost simultaneous, transmission from a CM as a parallel transmission. This abstraction is necessary due to the granularity of the simulation step in the model.

In hardware, a CM can transmit one or more messages almost simultaneously to the SMs. At most, these synchronization messages include one `CS` (coldstart), one `CA` (coldstart ack), and one `IN` (integration) for each *integration cycle*. That is, at most (number of integration cycles + 2) messages can be transmitted by a CM almost simultaneously. For background context on these messages, refer to the aforementioned protocol standard.

Although this abstraction simplifies the model, it does not capture the impact of non-concurrent reception of these messages by the SMs. In hardware, since these messages are transmitted sequentially, they are also received and processed sequentially by the SMs. Further, such a simultaneous transmission also leads to non-determinism in the reception order; due to the error term in the permanence points in time in TTEthernet. Different SMs may receive and process these messages in a different order. To accurately capture the behavior of TTEthernet startup protocol, this non-determinism must be explicitly captured in the SM state machine of the model.

In the model, the non-determinism in the reception order of `CA` or `CS` messages from different

Messages received	Constraints on priority	Message to be processed
only `IN`	$in_p > cs_p$ & $in_p > ca_p$	`IN`
only `CS`	$cs_p > ca_p$ & $cs_p > in_p$	-
only `CA`	$ca_p > in_p$ & $ca_p > cs_p$	`CA`
`IN` & `CS`	$in_p > ca_p \parallel cs_p > ca_p$	`IN`
`IN` & `CA`	$in_p > cs_p \parallel ca_p > cs_p$	`CA`
`CA` & `CS`	$ca_p > in_p \parallel cs_p > in_p$	`CA` or `CS`
`IN` & `CA` & `CS`	true	`CA` or `CS`

Table 1. Non-deterministic message reception in states that ignore `CS` messages

CMs need not be considered, because SMs process these messages independently of how many are received. The end result of processing is that one or more `CA` or `CS` messages depend only on the current state of the SM and does not depend on the number of such messages received. Our only concern is whether the SM received any `CA` or `CS` message at a particular instant, so we consider only the highest priority `CA` or `CS` message. For `IN` messages however, the number of messages received has an impact on the SM because it uses the membership vectors of all the received `IN` messages to make a decision. For example, if two SMs in the `SM_SYNC` state receive different subsets of `IN` messages, then, depending on the membership vectors of these messages, each SM can independently transition either to the `SM_STABLE` or the `SM_UNSYNC` state. Therefore, if completeness is desired, the non-determinism in the reception order of `IN` messages should be captured in the model. However, capturing this non-determinism would lead to an exponential blow-up, because it requires encoding all possible subsets of incoming `IN` messages—in the case of the presented model this would be 2^{60} (6 CMs and 10 integration cycles). We did not capture this non-determinism in `IN` messages; we assumed that all SMs receive only the highest priority `IN` messages.

States in the SM state machine can be classified into two types depending on how they react to `CS` messages: 1) states that ignore the `CS` message (`SM_INTEGRATE`, `SM_SYNC`, `SM_STABLE`, and `SM_TENTATIVE_SYNC`) and 2) states that process the `CS` message (`SM_UNSYNC`, `SM_FLOOD`, and `SM_-WAIT_4_CYCLE_START_CS`). For each of these state groups, the logic for modeling non-deterministic message reception is described below. Algorithm 1 presents the logic for states that ignore the `CS` message, and Algorithm 2 presents the logic for states that process the `CS` message. In these algorithms, the notation `a [ ] b` denotes a non-deterministic choice between actions `a` and `b`.

Line 3 in Algorithm 1 presents the action to be taken when incoming *transparent_messages* from CMs do not contain any `CS` or `CA` messages. In this case, the SM will process all the highest priority `IN` messages that were received. Lines 6- 8 present actions taken when the incoming message does not contain any `CA` message, but has a `CS` message. In this case, if it is also true that $cs_p > in_p$, then some SMs may not receive the highest priority `IN` messages. This situation occurs when the SMs first receive the higher priority `CS` message, which is ignored, and then their filters block all lower priority `IN` messages. Therefore, as shown in Line 6, we allow SMs to either process incoming `IN` messages or ignore them non-deterministically. On the other hand, if $in_p \geq cs_p$, then all SMs will receive the highest priority `IN` messages and will process them as shown in Line 8, ignoring any incoming `CS` messages. Similarly, Lines 14- 16 present actions for the case when the incoming message does not contain any `CS` message, but has a `CA` message. If a `CA` message is received, then the SM will ignore all `IN` messages and transition to the `SM_WAIT_4_CYCLE_START_CS` state. Therefore, when $in_p > ca_p$ some SM cannot receive any `CA` message and will process the

Algorithm 1 Non-deterministic message reception in states that ignore `CS` messages

```
Input  $transparent\_messages$        ▷ Set of messages received from CMs
Input  $cs_p$        ▷ Highest priority among all CS messages in $transparent\_messages$
Input  $ca_p$        ▷ Highest priority among all CA messages in $transparent\_messages$
Input  $in_p$        ▷ Highest priority among all IN messages in $transparent\_messages$
 1: if No CA message in $transparent\_messages$ then
 2:     if No CS message in $transparent\_messages$ then
 3:         Process IN messages with priority $in_p$.
 4:     else
 5:         if $cs_p > in_p$ then
 6:             Ignore all messages [ ] Process IN messages with priority $in_p$.
 7:         else
 8:             Process IN messages with priority $in_p$.
 9:         end if
10:     end if
11: else
12:     if No CS message in $transparent\_messages$ then
13:         if $in_p > ca_p$ then
14:             Process CA messages [ ] Process IN messages with priority $in_p$.
15:         else
16:             Process CA messages.
17:         end if
18:     else
19:         Process messages using Table 1.
20:     end if
21: end if
```

highest priority IN messages.

Finally, Line 19 represents the scenario when *transparent_messages* have both a CA and a CS message; Table 1 lists the messages to be processed for the message reception scenarios. For example, consider the scenario when an SM receives only a CA message, as shown in the third table row, which can happen when $ca_p > in_p$ and $ca_p > cs_p$. The corresponding action is to process the CA message. Likewise, the remaining lines in the table can be interpreted and encoded in the model. The resulting state machine has four boolean variables, one each for the four types of actions described in the algorithm. These variables encode the various scenarios under which those actions may be taken by some SM.

Algorithm 2, listed on page the next page, presents the non-determinism logic encoded in the model for the case when the SM is in a state that does not ignore the CS message. Since message processing is slightly different in each of the three relevant states (SM_UNSYNC, SM_FLOOD, and SM_WAIT_4_CYCLE_START_CS), they are presented separately in the algorithm.

Algorithm 2 Non-deterministic message reception in states that process CS messages

```
Input transparent_messages      ▷ Set of messages received from CMs
Input $cs_p$      ▷ Highest priority among all CS messages in transparent_messages
Input $ca_p$      ▷ Highest priority among all CA messages in transparent_messages
Input $in_p$      ▷ Highest priority among all IN messages in transparent_messages
 1: if No CA message in transparent_messages then
 2:     if No CS message in transparent_messages then
 3:         Process IN messages with priority $in_p$, if state is SM_UNSYNC.
 4:     else
 5:         if State is SM_UNSYNC then
 6:             Process CS messages [ ] Process IN messages with priority $in_p$.
 7:         else
 8:             if $cs_p \geq in_p$ then
 9:                 Process CS messages.
10:             else
11:                 Ignore all messages [ ] Process CS messages.
12:             end if
13:         end if
14:     end if
15: else
16:     if No CS message in transparent_messages then
17:         if $in_p > ca_p$ then
18:             Process CA messages [ ] Process IN messages with priority $in_p$, if state in SM_UNSYNC.
19:         else
20:             Process CA messages.
21:         end if
22:     else
23:         Process messages using Table 2.
24:     end if
25: end if
```

Messages received	Constraints on priority	Message to be processed		
		SM_UNSYNC	SM_FLOOD	SM_WAIT_4_CYCLE_START_CS
only IN	$in_p > cs_p$ & $in_p > ca_p$	IN	-	-
only CS	$cs_p > ca_p$ & $cs_p > in_p$	CS	CS	CS
only CA	$ca_p > in_p$ & $ca_p > cs_p$	CA	CA	CA
IN & CS	$in_p > ca_p \parallel cs_p > ca_p$	IN or CS	CS	CS
IN & CA	$in_p > cs_p \parallel ca_p > cs_p$	CA	CA	CA
CA & CS	$ca_p > in_p \parallel cs_p > in_p$	CS	CS	CS
IN & CA & CS	true	CS	CS	CS

Table 2. Non-deterministic message reception in states that process CS messages

3 Model Exploration and Scenario-based Test Generation

Once the model changes and extensions were implemented, we had to validate the revised model with respect to the proof attained for the initial model [3]. This section discusses the setup we used for this proof. In addition we present some addition scenario-based testing that we performed in order to validate the modified model's behavior. Before presenting the proof and test scenarios, we briefly discuss the various sal-atg options that were used in these experiments.

3.1 The sal-atg Tool

The sal-atg tool [4] is a relatively new member of the SAL tool suite from SRI that performs automated generation of efficient test sets using the method described in [5]. Automated test generation is used to construct a sequence of inputs that will cause the system under test to exhibit behaviors of interest—the test goals. The sal-atg tool generates test sequences from a SAL system specification that has been augmented with *trap variables* to encode the test goals. Trap variables are Boolean state variables that are initially FALSE and are set TRUE when some test goal is satisfied. For example, if the test goals are to achieve state and transition coverage, then each state and transition in the specification will have a trap variable associated with it; these will be set TRUE whenever their associated state or transition is encountered or taken. Further discussion on sal-atg can be found in the tool documentation on the SAL tool website [4].

Several options are available to seed the execution of sal-atg. Below, we discuss the options used in this work.

- `-s plingeling`: This option forces sal-atg to use `plingeling` [6] as the back-end SAT-solver for generating test cases. This is a very efficient solver, because it can use parallel threads to search the state-space. It is also possible to seed its execution with the number of desired parallel threads, which was very useful in our case. Since memory requirements of `plingeling` are directly proportional to the degree of parallelism, we limited the degree of parallelism to four in our runs.

- `-id` x: sal-atg uses this option to set the search depth used for the *initial test segments*; x is a non-negative integer. An initial test segment is a path in the model that originates from one of the initial states of the model. If a certain test goal can only be discharged at a depth of say 20, then to guarantee that sal-atg can reach this test goal, its execution must be seeded with the option `-id` 20.

- `-ed` x: With this option, sal-atg will search for *extension test segments* only up to a depth of x, where x is a non-negative integer. An extension test segment is a path in the model that originates from one of the initial test segments. If a certain test goal can be reached after some other test goal has been discharged, then we can use this option to ask sal-atg to discover this extension segment.

- `--branch`: This option forces sal-atg to explore extension segments only from the first initial segment that it discovered. It is important to understand that, when this option is specified, sal-atg will not explore other initial segments if it fails to find the extension segment from the first initial segment. Hence, we specify this option only when we know that the desired extension segment can be found from any initial test segment.

- `-v` x: This option specifies the level of verbosity desired from sal-atg, and it mainly helps in debugging the runs. We used $x = 4$ in our experiments and found this level to be sufficient for our purposes.

- `--fullpath`: With this option sal-atg outputs the test vectors with all the model variables, which can then be used to reproduce the tests on real hardware.

- `--incrext`: This option forces sal-atg to look for extension segments at incremental depths starting from 0 and going up to the value specified in `-ed`. It is an optimization to reduce the running time, so that test goals at shorter depths can be discharged faster.

In addition to the options listed above, we have also used two other options, namely `--latching` and `--noslice`, which are minor optimizations. Details of these options can be found in the tool documentation.

3.2 Model Validation using Proof

The main goal of a proof is to verify that the worst-case duration for the system to synchronize is less than 60 simulation steps. This finding was presented in the project report that contained the initial model [3]. We expect that the same proof holds for the extended model.

The model used for the proof has one priority level and assumes two faulty channels as described in Section 2.1. Since there is only one priority level, `CM2`, `CM4`, and `CM6`, from the system shown in Figure 1, are switched off. Note that switching off these CMs does not disconnect `CM1`, `CM3`, or `CM5`, from `SM4`, `SM5`, and `SM6`. The `sm_cm_connections` module ensures that messages are transmitted across these switched off CMs. This modeling choice simplifies the handling of scenarios with and without priorities and makes it possible to use the same model for both scenario types. Further, it is assumed that `CM1` and `CM2` can be non-deterministically faulty as described in Section 2.1, and these faults can change dynamically for each simulation step. Finally, neither the power on, sleep, and re-power on times of the SMs nor the power on time of `CM5` are constrained in any manner; therefore, they can be chosen non-deterministically as well. The faulty CMs (`CM1` and `CM3`) are powered on at start-up.

The model was first driven into a clique scenario in which at least two SMs disagreed on their clocks and are in either `SM_SYNC` or `SM_TENTATIVE_SYNC` state. The definition of this clique in the model is generic enough to allow all possible clique scenarios and ensure that the worst-case is explored by the proof. Once the model reaches this clique, the worst-case counter is started and counts the number of simulation steps until all the SMs are synchronized and are in `SM_SYNC` or `SM_STABLE` state.

The proof was executed using the *bounded model checker* `sal-bmc` [7], with Yices as the backend SAT-solver. The following command was used to execute the proof.

```
sal-bmc -v 4 -d 100 TTEthernet_startup_proof1 test > TTEthernet_startup_proof1.trc
2> TTEthernet_startup_proof1.stderr
```

A depth of 100 was considered for the proof because the worst-case counter is started only after the system reaches a clique scenario. This should be found at a depth of 35, however we wanted to ensure that the counter does not exceed 60. The results of the proof confirmed our expectation that the worst-case duration for the system to synchronize after the initial clique is less than 60 simulation steps. This check ensures that the extended model behaves consistently with the initial model, and that no extraneous behavior was introduced while incorporating priorities into the model.

3.3 Model Validation using Test Scenarios

Several test scenarios were generated for validating the behavior of the extended model. These scenarios explore different aspects of the protocol state machine, further increasing our confidence on the correctness of the model. Below we present these scenarios and discuss results.

1. **Periodic stability for high priority SM**: This test case demonstrates an interesting behavior of the TTEthernet startup protocol in the presence of priorities. Initially, all CMs and all the low priority SMs are powered on. Once the low-priority SMs synchronize their clocks and are in `SM_STABLE` state, one high-priority SM is powered on. The expected behavior from here on is that the low-priority SMs will continue to be in `SM_STABLE` state, while the high priority SM switches between `SM_SYNC` and `SM_INTEGRATE` or `SM_UNSYNC` states. This interesting behavior occurs because SMs are allowed to be in `SM_STABLE` state in the extended model even if they do not receive an `IN` message for one integration cycle. As a result, the low priority SMs continue to be in `SM_STABLE` state, while the high priority SM keeps fluctuating between the synchronized and unsynchronized states.

   Component priorities are fixed as shown in Figure 1. It is also assumed that `CM1`, `CM2`, `CM3`, and `CM4`, can be non-deterministically faulty as described in Section 2.1. However, to ensure portability of the resulting test vectors to hardware, the faults are restricted to be fixed at start-up. That is, sal-atg is free to choose a specific fault pattern for the earlier mentioned CMs at the beginning of execution, but it cannot change it dynamically. Initially `SM1`, `SM2`, and `SM3` are powered on, and once they synchronize, `SM4` is powered on.

   The following command was used to execute this test case, and sal-atg confirmed the expected behavior using one test at a depth of 38.

   ```
   sal-atg TTEthernet_startup_test_priority cluster TTEthernet_startup_test_priority.scm
   -s lingeling -id 30 -ed 30 -v 4 --branch --fullpath --incrext --latching --noslice
   > TTEthernet_startup_test_priority.trc 2> TTEthernet_startup_test_priority.stderr
   ```

2. **Low-priority clique, then high-priority swamp**: This test case demonstrates another interesting behavior of the TTEthernet startup protocol in the presence of priorities. Initially the low-priority SMs are allowed to form a clique of their own, without interference from the high-priority SMs. Once the clique is formed, the high- priority SMs can interact with the

low-priority SMs with the result that the high-priority SMs will swamp the low-priority SMs and force them to synchronize with the high-priority clock. An interesting observation in this test case was that, although the low-priority SMs synchronize to the high-priority time-line, their membership vectors are never updated to reflect this fact, because the low-priority `IN` messages in the scenario are always suppressed by the priority filters of all the SMs.

Component priorities and CM fault modeling are identical to the previous case. In this test case, we considered two different SM and CM power-on sequences. In the first case, only the low-priority SMs and all the CMs are powered on initially, and once the low-priority SMs form a clique, the high-priority SMs are powered on. In the second case, all the SMs are powered on at start-up, and to enable low priority clique formation, the non-faulty CMs (`CM5` and `CM6`) are powered on only after the low priority SMs synchronize with each other.

The following command was used to execute this test scenario with the first SM and CM power on sequence described above. The sal-atg tool confirmed the expected behavior using one test at a depth of 58.

```
sal-atg TTEthernet_startup_test_pty1 cluster TTEthernet_startup_test_pty1.scm
-s lingeling -id 35 -ed 30 -v 4 --branch --fullpath --incrext --latching --noslice
> TTEthernet_startup_test_pty1.trc 2> TTEthernet_startup_test_pty1.stderr
```

The following command was used to execute this test scenario with the second SM and CM power on sequence described above. The sal-atg tool confirmed the expected behavior using one test at a depth of 38.

```
sal-atg TTEthernet_startup_test_pty2 cluster TTEthernet_startup_test_pty2.scm
-s lingeling -id 35 -ed 30 -v 4 --branch --fullpath --incrext --latching --noslice
> TTEthernet_startup_test_pty2.trc 2> TTEthernet_startup_test_pty2.stderr
```

3. **Swinging membership**: This test case demonstrates a swinging membership pattern for the extended model with one priority level. This pattern includes the following steps: 1) All the SMs are powered on and allowed to synchronize their clocks; 2) Three SMs (`SM3`, `SM4`, and `SM5`) are switched off, leaving the other three to remain synchronized; 3) The switched off SMs are re-powered on and allowed to synchronize with the other SMs. Since the test case uses only one priority level, `CM2`, `CM4`, and `CM6` are switched off, and the remaining CMs are powered on at startup. The fault model is identical to the first test case on "Periodic stability for high priority SM."

   The following command was used to execute this test scenario, and sal-atg confirmed the expected behavior using one test at a depth of 53.

```
sal-atg TTEthernet_startup_test_swinging cluster TTEthernet_startup_test_swinging.scm
-s lingeling -id 35 -ed 30 -v 4 --branch --fullpath --incrext --latching --noslice
> TTEthernet_startup_test_swinging.trc 2> TTEthernet_startup_test_swinging.stderr
```

4. **Incremental membership**: This test case demonstrates an incremental membership pattern for the extended model with one priority level. This pattern uses the following steps:

1) Three SMs (`SM1`, `SM2` and `SM4`) are powered on and allowed to synchronize their clocks; 2) The remaining SMs are powered on and allowed to synchronize with the other SMs, one at a time. That is, `SM3` is powered on and allowed to synchronize, then `SM5` is powered on and allowed to synchronize, and finally `SM6` is powered on. Since the test case uses only one priority level, `CM2`, `CM4`, and `CM6` are switched off, and the remaining CMs are powered on at startup. The fault model is identical to the first test case on "Periodic stability for high priority SM."

The following command was used to execute this test scenario, and sal-atg confirmed the expected behavior using 1 test at a depth of 78.

```
sal-atg TTEthernet_startup_test_incremental cluster TTEthernet_startup_test_incremental.scm
-s lingeling -id 80 -ed 0 -v 4 --fullpath --latching --noslice
> TTEthernet_startup_test_incremental.trc 2> TTEthernet_startup_test_incremental.stderr
```

4 Coverage-based Test Generation

In this section, we describe test generation for achieving protocol decision logic coverage. The main goal of this test generation is to achieve (MC/DC) coverage of the TTEthernet startup protocol, via normal protocol action constrained within a valid fault-scenario with respect to the protocol assumptions. We allowed for two faulty CMs to be present within a three-channel system, as described in Section 2.1. We consider both high- and standard- integrity models.

To ease the transfer of the generated test cases to hardware, we applied a fault model restricted to permanent fault scenarios; that is, connection failures are held consistent throughout the entire test scenario. The rationale for this is twofold: 1) If coverage is achieved with this restrictive fault model, we can be assured that coverage will be attainable with a more relaxed fault model as well; 2) A permanent fault model may simplify the execution of the tests on the TTEthernet validation hardware test bed since it removes the need to synchronize fault switching to the protocol execution flow.

Besides the permanent fault injection on two CMs, the test generation procedure can also dynamically power on or put to sleep SMs and can delay the power on of the non-faulty CM. This additional level of control is aligned with the capability of the TTEthernet hardware validation test bed.

4.1 Model Instrumentation

To satisfy the MC/DC coverage criterion, all of the following conditions had to be observed at least once during the test campaign [8].

1. Each decision tries every possible outcome.
2. Each condition in a decision takes on every possible outcome.
3. Each entry and exit point is invoked.
4. Each condition in a decision is shown to independently affect the outcome of the decision. Independence is shown by changing one condition at a time and observing its impact on the decision outcome.

The first step of MC/DC test coverage generation is to instrument the model. We achieved this by adding trap variables to every transition of the SM/CM/filter state machines. Further, when a transition guard used logical `OR`, additional trap variables were introduced to observe the independent impact of each condition in the guard. Note that no trap variables were introduced for the non-deterministic transitions described in Section 2.2, because the non-deterministic transitions are only a consequence of the modeling abstraction and are not part of the startup protocol itself.

To understand the coverage instrumentation, consider the code-snippet for the high integrity model shown below. It represents the transition from `SM_SYNC` to `SM_UNSYNC` state in the SM state machine. The guard `(cs_ignore_states_ignore OR cs_ignore_states_in)` models the non-determinism described in Section 2.2, so we do not consider it for MC/DC coverage. As indicated by the rest of the transition guard, two conditions participate in a logical `OR`: condition `mem2nat(SM_local_async_membership) >= sm_sync_to_unsync_threshold_async` and condition `mem2nat(SM_local_async_membership) >= mem2nat(SM_local_sync_membership)`. The first condition tests whether the asynchronous membership vector is above the high integrity threshold for `SM_SYNC` state, while the second tests whether the asynchronous membership vector is at least as big as the synchronous membership vector. In either case, the SM must transition to the `SM_UNSYNC` state. To achieve MC/DC coverage of this transition, we consider three trap variables: `trap_30b`, `trap_30c`, and `trap_30d`, shown below. Together, these variables ensure that the independent impact of each condition on this transition is evaluated.

```
[]
  SM_state = SM_SYNC AND
  (cs_ignore_states_ignore OR cs_ignore_states_in) AND
  inctime(SM_local_clock) = sm_dispatch_int AND
  (mem2nat(SM_local_async_membership) >= sm_sync_to_unsync_threshold_async OR
  mem2nat(SM_local_async_membership) >= mem2nat(SM_local_sync_membership)) AND
  NOT sm_sleep_timeout
-->
  trap_30b' = IF mem2nat(SM_local_async_membership) >= sm_sync_to_unsync_threshold_async
              AND mem2nat(SM_local_async_membership) < mem2nat(SM_local_sync_membership)
              THEN TRUE ELSE FALSE ENDIF;
  trap_30c' = IF mem2nat(SM_local_async_membership) >= mem2nat(SM_local_sync_membership)
              AND mem2nat(SM_local_async_membership) < sm_sync_to_unsync_threshold_async
              THEN TRUE ELSE FALSE ENDIF;
  trap_30d' = IF mem2nat(SM_local_async_membership) >= mem2nat(SM_local_sync_membership)
              AND mem2nat(SM_local_async_membership) >= sm_sync_to_unsync_threshold_async
              THEN TRUE ELSE FALSE ENDIF;
  SM_state' = SM_UNSYNC;
```

Once the model was instrumented, attaining coverage was surprisingly straightforward using sal-atg. The trap variables of interest were simply entered as test goals. With respect to coverage, our first idea was to evaluate the coverage attained from two perspectives. One run, *_allobservers*, would have an unconstrained observation point, where coverage could be achieved by any one of the six state machine instances firing a trap variable. The *_oneobserver* run would have a constrained observation point, where we required a specific state machine instance to fire a trap variable. However, our attempt at instrumenting the all-observer case missed an unanticipated behavior of sal-atg. When a trap variable is defined locally for each instance of the state machines, sal-atg *does not* automatically bind the test goal for that trap variable to the logical `OR` of all of the local variable instances; instead it selects only one instance for monitoring. Hence, in practice our *_allobservers* test runs were not fully-specified instances of *_oneobserver* test cases. In addition, since the *_oneobserver* runs encountered no difficulty in attaining full coverage with the limited time available (see Section 4.3), we focused only on the *one_observer* test runs with respect to test vector analysis.

4.2 Instrumented Model File Description

4.2.1 TTEthernet_startup_coverage_sm_cm_one(all)observer(s)

These files were used to attain coverage of the SM and CM state machines for the high-integrity case. Since priorities have no direct impact on the SM or CM state machines, all of the SMs and CMs operate at a single priority level for these tests. Because only one priority level is used, `CM2`, `CM4`, and `CM6` are switched off from the system shown in Figure 1. Note that switching off these CMs does not disconnect `CM1`, `CM3`, or `CM5` from `SM4`, `SM5`, and `SM6`. The `sm_cm_connections` module ensures that messages are transmitted across these switched off CMs. In accordance with the fault generation hypotheses, the coverage runs `CM1` and `CM2` were configured to be non-deterministically faulty, relaying and receiving only a subset of the messages. Once again, the faults of `CM1` and `CM2` were assumed to be permanent to ease porting the resulting test vectors to the TTEthernet hardware validation test bed. That is, sal-atg can choose a specific fault pattern for `CM1` and `CM3` at the beginning of execution but cannot change it dynamically cycle by cycle. The coverage models do not constrain the power on, sleep, and re-power on times of the SMs, nor the power on time of `CM5`, the fault-free CM. The faulty CMs—`CM1` and `CM3`—are powered on at the beginning of the scenario. `SM1` was used as the observer for coverage of the SM state machine, and the non-faulty `CM5` was used as the observer for coverage of the CM state machine. The SAL model corresponding to the one observer case is presented is posted on the NASA DASH*link* AFCS-Distributed Systems site (see Appendix A). For the standard integrity case, the model used to attain coverage of the SM and CM state machines is in the file TTEthernet_startup_coverage_sm_cm_one_observer_si.

4.2.2 TTEthernet_startup_coverage_filter_one(all)_observer(s)

These files were used to attain coverage of the priority filter state machine for the high-integrity case. For this test, the default priorities were unspecified for the SMs and CMs, enabling sal-atg to non-deterministically choose values at the beginning of the test scenario. Since two priority levels are required to attain this coverage, all CMs are powered up at the start of the test scenario. The fault specification is consistent with that of the sm_cm_coverage models, in that all the CMs in `channel 1` and `channel 2` can non-deterministically omit messages. To ensure portability of the resulting test vectors to the hardware test bed, the connectivity is assumed to be permanent and fixed at start-up. Similar to the sm_cm_coverage runs, the filter coverage models do not constrain the power on, sleep, and re-power on times of the SMs, nor the power on time of `CM5` and `CM6`, the fault-free CMs. The faulty CMs—`CM1`, `CM2`, `CM3`, and `CM4`—are powered on at the beginning of the scenario. The priority filter of `SM1` was used as the observer for coverage in this case. The SAL model corresponding to the one observer case is posted on the NASA DASH*link* AFCS-Distributed Systems site (see Appendix A). For the standard integrity case, the model used to attain coverage of the priority filter state machine is in the file TTEthernet_startup_coverage_filter_one_observer_si.

4.3 Test Coverage Results

4.3.1 Summary

The performance of sal-atg in conjunction with Lingeling SAT-solver was impressive. For all of the coverage models, tests achieved full coverage of the reachable state transitions. For both the high-integrity and standard-integrity configurations, the SM coverage runs completed in approximately two days of execution time using three processing cores. The CM and priority filter coverage runs completed in less than a day of execution using four processing cores.

4.3.2 Detailed findings

The sal-atg tool ran into memory problems (heap space allocations) when we tried to run it with 54 trap variables for coverage of the SM state machine. We suspect this issue occurred because sal-atg has to record the test vectors for all the discharged trap variables at any point in time. This problem was resolved by splitting the trap variables into five sets and running sal-atg on each set independently. By splitting variables into groups, sal-atg is forced to generate separate test vectors for discharging each group of variables, which can increase the number of generated tests. We were able to reduce the number of tests by using information from the failed runs. Any two variables that were discharged by the same test vector in the failed runs were kept in the same group. With this strategy, we were able to discharge 50 of the 54 trap variables for coverage of the SM state machine. The four undischarged variables are not reachable in the high-integrity configuration; these are discussed in Section 4.3.3.

As described earlier, we performed two experiments for each coverage requirement. One in which all the SMs and CMs act as observers for discharging the test goals, and the other in which exactly one SM or CM is used to discharge the test goals. An interesting observation from these experiments is that in both cases, sal-atg produced identical test scenarios. That is, even when we did not restrict the observer to a single SM or CM, sal-atg only used `SM1` or `CM1` to discharge the test goals.

It is also worth noting that although the resulting test vectors were identical, there was a significant difference in the running time of sal-atg for the two cases. The case with one observer was markedly slower than the case with all observers. We suspect the main reason for this difference is that the one-observer model uses global trap variable arrays instead of the local trap variables as in the all-observers model. Note that this observation must *not* be relied upon as a matter of fact, because the two experiments were performed on similar, but different machines.

4.3.3 SM coverage test results

We used sal-atg in two different modes to achieve coverage on the SM state machine. For the first two trap variable groups, we allowed sal-atg to explore extension test segments once a trap variable was discharged by some initial test segment. For the remaining three trap variable groups, we did not allow sal-atg to explore extension segments. We based this decision on our observations in the failed runs. We did not observe any variable being discharged by extension segments in the last three groups. So to reduce the running time of sal-atg, we disabled extension segments for these variable groups.

These five runs for the high-integrity configuration are archived under the sm_runs directory as traces TTEthernet_startup_coverage_sm_oneobserver_first.trc, TTEthernet_startup_coverage_sm_oneobserver_second.trc, TTEthernet_startup_coverage_sm_oneobserver_third.trc, TTEthernet_startup_coverage_sm_oneobserver_fourth.trc, and TTEthernet_startup_coverage_sm_oneobserver_fifth.trc. For the standard integrity configuration, the runs are archived under the sm_runs directory as traces TTEthernet_startup_coverage_sm_oneobserver_si_first.trc, TTEthernet_startup_coverage_sm_oneobserver_si_second.trc, TTEthernet_startup_coverage_sm_oneobserver_si_third.trc, TTEthernet_startup_coverage_sm_oneobserver_si_fourth.trc, and TTEthernet_startup_coverage_sm_oneobserver_si_fifth.trc.

For coverage of the SM state machine, sal-atg did not discharge four trap variables in the high-integrity case -*trap_8*, *trap_13*, *trap_23*, and *trap_30a*. Below, we present the transitions corresponding to these variables and justify the lack of coverage.

1. **Transition from `SM_UNSYNC` to `SM_TENTATIVE_SYNC`**: This transition requires a membership vector of 6 in the high integrity case. In the system with six SMs, if one SM is in the `SM_UNSYNC` state, then at most five SMs can be synchronized; hence, the membership vector can have no more than five SMs. Therefore, this transition can never be enabled in our test configuration.

2. **Self-loop transition in `SM_TENTATIVE_SYNC` for sending `IN` messages**: Once an SM enters `SM_TENTATIVE_SYNC` state, it immediately transitions to either `SM_SYNC` or `SM_UNSYNC` in the next simulation step. The `SM_TENTATIVE_SYNC` state is thus a transient state, and no SM can execute the self-loop transition.

3. **Self-loop transition in `SM_FLOOD` for decreasing the timer in the step immediately**

after entry into the state: This transition is not feasible in the high-integrity case, because the timer is always set to zero upon entry into the state.

4. **Transition from `SM_SYNC` to `SM_UNSYNC` upon reception of `IN` message**: This transition is not feasible in the high-integrity case, because the membership vector required for this transition is a non-integer between 0 and 1.

In the standard-integrity configuration, sal-atg did not discharge five trap variables— *trap_8*, *trap_13*, *trap_23*, *trap_30c*, and *trap_35c*. Note that, except for the threshold values, the SM state machine in the standard-integrity configuration is identical to that in the high-integrity configuration. Below, we present the transitions corresponding to these undischarged variables, and justify the lack of coverage.

1. The reasoning for the lack of coverage of trap variables *trap_8*, *trap_13*, and *trap_23*, is identical to the high-integrity case.

2. Variable *trap_30c* corresponds to the transition from `SM_SYNC` to `SM_UNSYNC` based on the size of the asynchronous membership vector. This transition is not feasible in the standard-integrity case, because the asynchronous membership vector required for this transition must be smaller than 1 and, at the same time, greater than the synchronous membership vector.

3. Variable *trap_35c* corresponds to the transition from `SM_STABLE` to `SM_INTEGRATE` based on the size of the asynchronous membership vector. This transition is not feasible in the standard-integrity case, because the asynchronous membership vector required for this transition must be smaller than 1 and, at the same time, greater than the synchronous membership vector.

4.3.4 CM coverage test results

The CM coverage tests completed with similar success to the SM tests. The initial test run for the high-integrity configuration, executed with an initial search depth of 30, discharged all but two of the trap variables. This run is archived under the cm_runs directory as TTEthernet_startup_coverage_cm_oneobserver.trc. The two variables not discharged in this initial run were *trap_cm_9b* and *trap_cm_9d*. The variable *trap_9b* was later discharged by extending the initial search depth to 38. This run is archived as TTEthernet_startup_coverage_cm_tentative_oneobserver.trc. For the variable *trap_9d*, we concluded that it could not be discharged by a non-faulty CM, if all SMs were also non-faulty, as explained in the *trap_9d* trap variable logic given below.

```
IF mem2nat(CM_local_async_membership) < cm_tentative_to_unsync_threshold_async
   AND
   mem2nat(next_message_out[1][CM_local_integration_cycle].membership_new)
   < cm_tentative_to_sync_threshold
   AND
   mem2nat(next_message_out[1][CM_local_integration_cycle].membership_new)
   >= cm_tentative_to_unsync_threshold_sync
THEN
   TRUE
ELSE
   FALSE
ENDIF;
```

To exercise this trap variable, the CM must remain in `CM_TENTATIVE_SYNC` state for at least one integration cycle and also detect a persistent membership vector of size exactly 2 during this time.

However, non-faulty SMs will not continue to execute to support this scenario. That is, if only two SMs have synchronized their clocks, the SM cluster will not persist for a duration of one integration cycle and will immediately transition to the `SM_UNSYNC` state. As a result of this situation, the non-faulty CM will not be able to exercise this transition in the `CM_TENTATIVE_SYNC` state. However, because it can listen to a subset of powered-on SMs, a faulty CM may be able to exercise this transition. We observed this transition in the dedicated test run executed for this purpose. In this run, one of the faulty CMs exercised this transition at a depth of 38. The corresponding trace file is archived as TTEthernet_startup_coverage_cm_tentative_oneobserver_2.trc. In this case, although more than two SMs had synchronized their clocks, the faulty CM was receiving messages from only two of them and able to detect a persistent membership vector of size exactly 2.

For the standard integrity configuration, all but seven trap variables were discharged by a single run executed with a search depth of 38. Below, we present the transitions corresponding to the seven undischarged variables and justify the lack of coverage.

1. **Transition from `CM_UNSYNC` state to `CM_TENTATIVE_SYNC` state**: This transition is infeasible in the standard-integrity configuration, because it requires a membership vector smaller than 3 and at the same time larger than or equal to the number of SMs in the system, which in our case is 6.

2. **Trap variables in the `CM_TENTATIVE_SYNC` state**: In the standard-integrity configuration, state `CM_TENTATIVE_SYNC` is unreachable, because of the infeasibility of the transition from `CM_UNSYNC` to `CM_TENTATIVE_SYNC`. Hence, none of the trap variables for transitions originating from the `CM_TENTATIVE_SYNC` state were discharged.

3. **Transition from `CM_INTEGRATE` state to `CM_WAIT_4_CYCLE_START` state**: This transition is also infeasible in the standard-integrity configuration, because it requires a membership vector smaller than 3 and at the same time larger than or equal to the number of SMs in the system, which in our case is 6.

4.3.5 Priority filter coverage test results

The filter coverage test completed with full coverage for both the high-integrity and standard-integrity configurations. These runs are archived under the filter_runs directory as TTEthernet_-startup_coverage_filter_oneobserver.trc and TTEthernet_startup_coverage_filter_oneobserver_si.trc, respectively.

4.4 Example Test Generation Output

In this section we present two interesting test scenarios from the high-integrity test cases, one from the CM coverage tests and another from the SM coverage tests.

4.4.1 SM coverage test scenario

For the SM coverage tests, we present the test scenario corresponding to trap variable `trap_-30c` in the SAL model is posted on the NASA DASH*link* AFCS-Distributed Systems site (see Appendix A). This scenario corresponds to the transition from state `SM_SYNC` to state `SM_UNSYNC` when the asynchronous membership count exceeds the synchronous membership count in the SM state machine.

The connectivity diagram for this scenario is shown in Figure 3. Initially, CM1 and CM3 are switched-on and CM2, CM4, CM6, and CM5 are all switched-off. Since this test scenario uses only

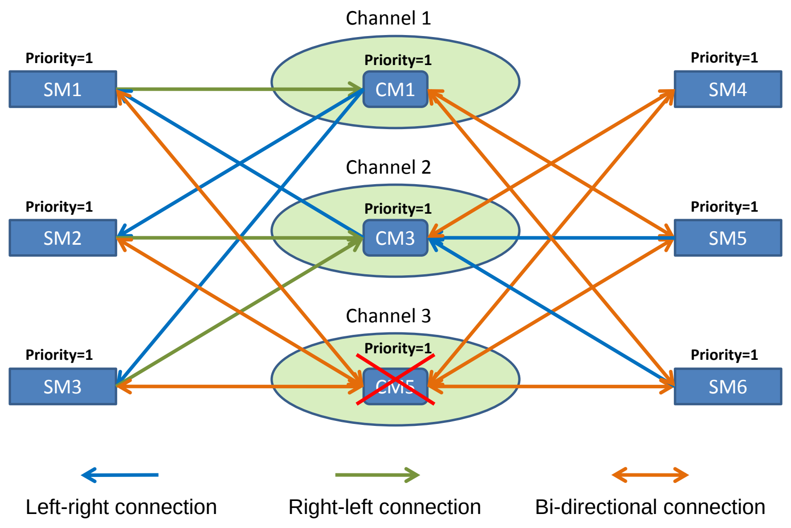

Figure 3. Connectivity diagram for SM test scenario (trap variable 30c)

a single priority level, CM2, CM4, and CM6 are never switched-on. CM5, as shown in Figure 4, is switched-on at Step 20 of the test trace. Since CM5 is a non-faulty CM, all connections between CM5 and the SMs are good. On the other hand, CM1 and CM3 are faulty, and therefore their connections with the SMs are not all good, as can be seen in Figure 3. For instance, CM1 can receive messages only from SM1, SM5, and SM6, and can send messages only to SM2, SM3, SM5, and SM6. It is completely disconnected from SM4. Note that this connectivity is determined at initialization by `sal-atg` and remains fixed for the duration of the test trace.

The test trace for trap variable `trap_30c` is shown in Figure 4. This trace is a sanitized version of the trace output generated by `sal-atg`, in which unnecessary SM and CM sleep and power-on events are eliminated. In the traces generated by `sal-atg`, several sleep and power-on events have no impact whatever on the trace outcome. We eliminated such events to simplify the traces. In this figure, `Sx` denotes SMx and `Cx` denotes CMx. The SM/CM list on top of the step-number line denotes power-on events for the respective SMs and CMs. Likewise, the SM/CM list below the step-number line denotes sleep events for SMs and CMs. The boxes provide system information such as component states and membership vectors at various stages of the test.

In Step 20 SM1 sends a `CA` message and moves to the `SM_FLOOD` state, while in Step 21 SM5 and SM6 send `IN` messages and move to the `SM_TENTATIVE_SYNC` state. In Step 22 SM1 moves back to `SM_UNSYNC` state as it does not receive its own `CA` message, because SM1 can send messages only to CM1 and CM5 and receive messages only from CM3 and CM5. CM5, although powered on, is not yet fully operational. Also in Step 22, SM3 and SM5 receive the `CA` message from SM1 and move to `SM_WAIT_FOR_CYCLE_START_CS` state. In Step 23 SM1 receives the `IN` message from SM5 and SM6 and moves to `SM_SYNC` state to synchronize with SM5 and SM6. It is interesting to note that at this stage SM3 and SM5 are synchronizing with SM1, whereas SM1 itself has already synchronized with the old clock of SM5 and SM6. This scenario is basically a consequence of the asymmetric connectivity in the system. In Step 26, after SM5 and SM6 have been forced into sleep, SM3 sends `IN` message and moves to `SM_TENTATIVE_SYNC` state. In Step 28, after receiving the `IN` message

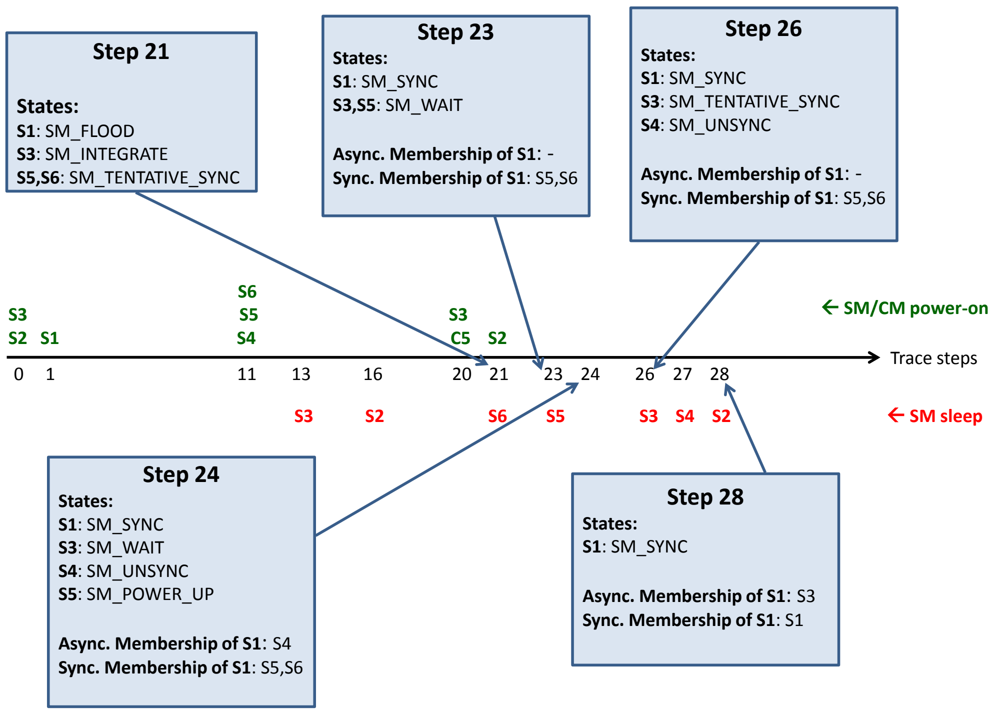

Figure 4. Execution trace for SM test scenario (trap variable 30c)

from SM3, the synchronous membership of SM1 has a size of 1 (itself only) and the asynchronous membership of SM1 also has a size of 1 (SM3). Therefore, at the start of the next integration round in Step 31, SM1 executes the transition with trap variable `trap_30c` to move from `SM_SYNC` to `SM_UNSYNC` state. Note that CM5 did not play any role in enabling the target transition, and the entire scenario was only feasible because of the asymmetric connectivity between SMs and the faulty CMs.

4.4.2 CM coverage test scenario

For CM coverage, we considered the test scenario corresponding to trap variable `trap_cm_9d` in the SAL model posted on the NASA DASH*link* AFCS-Distributed Systems site. (see Appendix A). This scenario corresponds to the self-loop transition in state `CM_TENTATIVE_SYNC`, which is taken only when the asynchronous membership count is less than 2 and the synchronous membership count is less than 3.

The connectivity diagram for this scenario is shown in Figure 5. Similar to the SM test scenario, CM2, CM4, and CM6 are switched-off, and CM1 and CM3 are switched-on throughout the test. CM5 is initially switched-off, but is later switched-on during Step 22 of the test trace as shown in Figure 6. CM5 has perfect connections to all the SMs, because it is non-faulty; whereas, CM1 and CM3 have inconsistent connectivity. For instance, CM3 can receive messages only from SM1, SM4, SM5, and SM6 and can send messages only to SM2, SM3, SM4, and SM5. Similar to the SM test scenario, this connectivity is determined at initialization by `sal-atg` and remains fixed for the duration of the test trace.

The test trace for trap variable `trap_cm_9d` is shown in Figure 6. This trace is a sanitized version of the trace output generated by `sal-atg`, in which unnecessary SM and CM sleep and

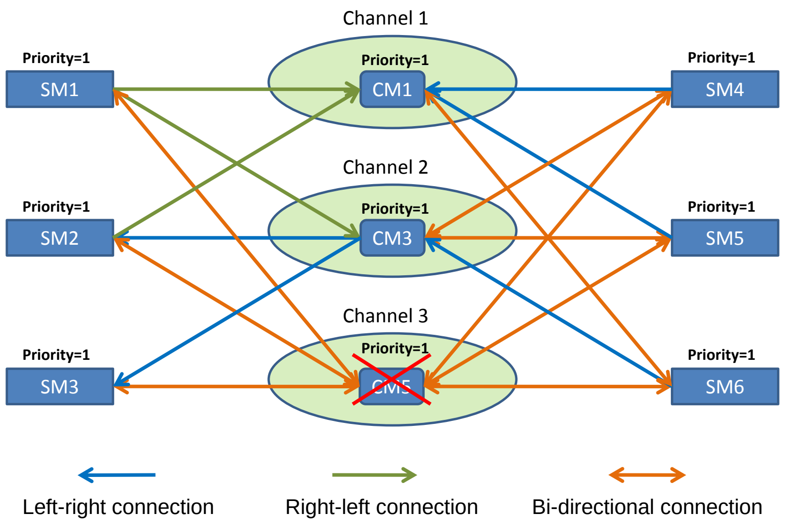

Figure 5. Connectivity diagram for CM test scenario (trap variable 9d)

power-on events are eliminated. In Step 24, SM3, SM4, and SM5 send `IN` messages and move to `SM_-TENTATIVE_SYNC` state. Upon receiving these `IN` messages in Step 25, CM1 moves from `CM_UNSYNC` to `CM_TENTATIVE_SYNC` state. Since CM1 cannot receive messages from SM3, it receives only two `IN` messages and moves to `CM_TENTATIVE_SYNC` instead of `CM_SYNC`. At the same time, CM5, which was powered on in Step 22, receives all 3 `IN` messages and moves to `CM_SYNC` state. In Step 26, upon receiving the `IN` messages, SM2 moves from `SM_UNSYNC` to `SM_SYNC` state to synchronize with SM5. SM3 and SM4 are forced into sleep at this stage. In Step 29, after completing one integration round, SM2 and SM5 send `IN` messages for the next round. In Step 30, CM1 receives these `IN` messages and executes the self-loop transition in `CM_TENTATIVE_SYNC` state corresponding to trap variable `trap_cm_9d`. This is because the synchronous membership size of CM1 is 2 and its asynchronous membership size is 0 at this stage.

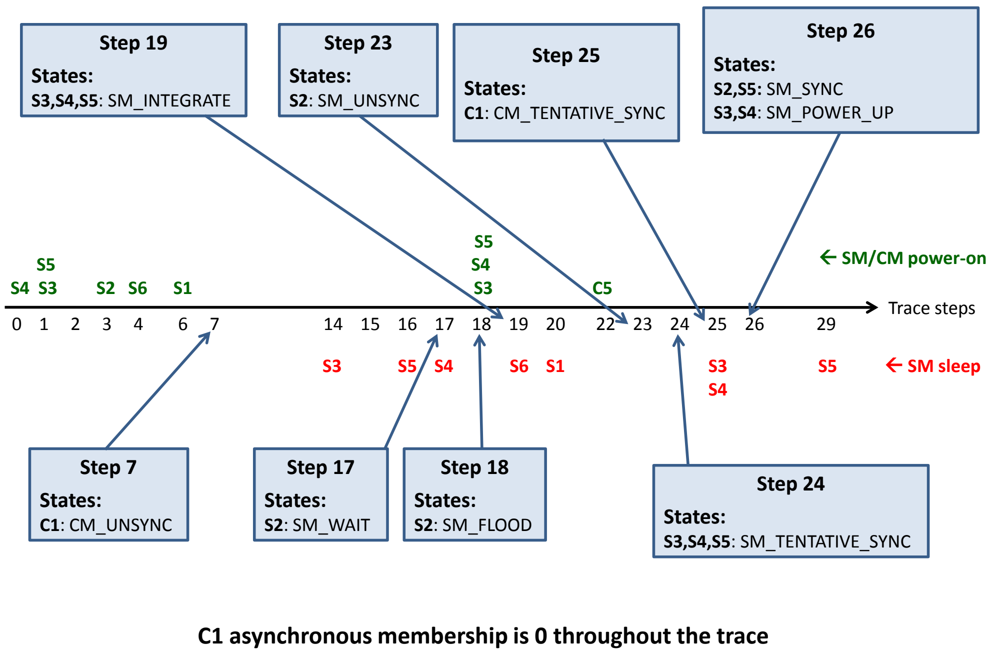

Figure 6. Execution trace for CM test scenario (trap variable 9d)

5 Conclusions and Future Work

Modern desktop computer hardware has reached a stage where formal methods technology is feasible for distributed test generation. The techniques are memory intensive, but in recent years, memory requirements have become accessible for normal desktop hardware. In our experiments, we demonstrated sal-atg's ability to generate tests that yield full MC/DC coverage with minimal human guidance. The performance of Lingeling and Plingeling sat-solvers also reduced the test generation duration by a factor of 4-5 relative to Yices. When test generation takes half a day to two days, such improvements can be critical to the real-world applicability of these new test generation methods.

Given the memory-intensive nature of the techniques explored, a common frustration encountered during this work was the erroneous termination of test generation procedures due to memory availability or heap management issues. In some cases, failure occurred after a day of execution and a run terminated with no results. The new capabilities of sal-atg2, added by Bruno Dutertre of SRI as part of this work package, were of great assistance. The sal-atg2 tool records the test vectors as test goals are discharged, so that even in erroneous scenarios, successful test vectors are salvageable.

During this study, we manually augmented the model to instrument MC/DC coverage. However, we believe that automatic augmentation may be a useful simple extension to the SAL-tool suite and would provide a less error-prone, systematic application of the techniques explored herein.

The test scenarios presented in this study, have been generated with the capability of the TTEthernet hardware validation test bed in mind. The only test orchestration required to execute test scenarios is limited to static connectivity mappings and the sequencing of end-system and switch power on and power off operations. Hence transferring test tests onto the hardware test platform is anticipated to be relatively straight forward. The only real difficulty is the alignment and treatment of time. With respect to clock drift this is also mitigated by the hardware test bed capability to use a common clock. With respect to TDMA phase alignment, the situation is a little more complicated. Unfortunately, the TTEthernet hardware requires a significant time to initialize from power on, and this time was not accommodated within the test generation models. Therefore this time needs to be accommodated within the hardware scenarios. For the single observer case this should be simple to achieve by aligning the phase of the completion of the configuration, with respect to the unit under test to the operating phase of the test stimulus.[3].

A second issue of the hardware test bed is the limited visibility of protocol flow. As part of the phase 2 work we shall execute the test scenarios within an instrumented simulation test bench to allow the formal comparison of VHDL code coverage with the predicted coverage of the generated tests.

We believe that we have been successful in reaching the original goals of this work package. The generation of MC/DC coverage tests from system-derived scenarios exceeded our expectations, and we achieved full coverage of the high-integrity protocol action for both end-system (SM) and switch (CM) components. Through informal exploration test development, we also developed a good understanding of how best to abstract target scenarios. At the time of writing this report, we admit that further effort is required to hide the complexities of SAL to a novice user base.

The integration of these techniques with emerging state-of-the-art techniques such as constrained, random testing may be a promising avenue of exploration.

[3] Achieved by a simple phase offset

References

1. AS-2D2 Embedded Computing Systems Committee SAE: Time-Triggered EThernet. SAE Standards N^o AS6802A, June 2011.

2. Bhatt, D.; Madl, G.; Oglesby, D.; and Schloegel, K.: Towards Scalable Verification of Commercial Avionics Software. *Proceedings of the AIAA Infotech@Aerospace Conference*, April 2010.

3. Steiner, W.: TTEthernet Executable Formal Specification. Complexity Management for Mixed-Criticality Systems (CoMMiCS). *Contact tta-group.com*, 2010.

4. SAL Automated Test Generator (sal-atg). http://www.csl.sri.com/users/rushby/abstracts/sal-atg.

5. Hamon, G.; de Moura, L.; and Rushby, J.: Generating Efficient Test Sets with a Model Checker. *In 2nd International Conference on Software Engineering and Formal Methods*, 2004, pp. 261–270.

6. SAT-solver Lingeling/Plingeling. http://fmv.jku.at/lingeling/.

7. SAL Bounded Model Checker (sal-bmc). http://sal.csl.sri.com/introduction.shtml.

8. Certification Authorities Software Team (CAST) Paper: CAST–10. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_-papers/media/cast-10.pdf.

Appendix A

TTEthernet Startup Protocol SAL Models and Results

The AFCS-Distributed Systems project has been established on NASAs DASH*link* to support public dissemination of the models and results of this program. The URL https://c3.nasa.gov/dashlink/ will take the user to the Home Page of DASH*link*. The user can access the site by hovering over the RESEARCH AREAS pull-down list and right clicking on Verification and Validation. Scroll down to AFCS Distributed Systems and right click the project. Right click TTEthernet SAL Models under Popular Resources and several source files are identified.

The zip file, TTEthernetStartupProtocolSALModelsAndResults.zip, contains a number of text files.

- Instructions are provided in README.txt
- SAL models are provided in *.sal files
- Scheme source files are provided in *.scm files
- Trace files are provided in *.trc files. If a test vector is found, then it is shown in the .trc file
- Results are provided in the *.stderr files

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

1. REPORT DATE *(DD-MM-YYYY)* 01- 09 - 2012	**2. REPORT TYPE** Contractor Report	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Model-Driven Test Generation of Distributed Systems	**5a. CONTRACT NUMBER** NNL10AB32T
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Easwaran, Arvind; Hall, Brendan; Schweiker, Kevin	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER 534723.02.02.07.30
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, Virginia 23681-2199	**8. PERFORMING ORGANIZATION REPORT NUMBER**
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/CR-2012-217764

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

Langley Technical Monitor: Paul S. Miner

14. ABSTRACT

This report describes a novel test generation technique for distributed systems. Utilizing formal models and formal verification tools, specifically the Symbolic Analysis Laboratory (SAL) tool-suite from SRI, we present techniques to generate concurrent test vectors for distributed systems. These are initially explored within an informal test validation context and later extended to achieve full MC/DC coverage of the TTEthernet protocol operating within a system-centric context.

15. SUBJECT TERMS

Clock synchronization; Fault tolerance; Formal methods; MC/DC coverage; Test generation

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)
					19b. TELEPHONE NUMBER ***(Include area code)***
U	U	U	UU	35	(443) 757-5802

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18