MODEL-BASED ANALYSIS OF TIMED-TRIGGERED ETHERNET

Bruno Dutertre, SRI International, Menlo Park, CA

Arvind Easwaran, Brendan Hall, Honeywell International, Minneapolis, MN

Wilfried Steiner, TTTech Computertechnik AG, Vienna, Austria

Abstract

Timed-Triggered Ethernet (TTEthernet) is a communication infrastructure that enables the use of Ethernet networks in real-time, distributed applications. The core of TTEthernet is a set of fault-tolerant protocols for clock synchronization, startup, and clique detection and resolution. We present recent work on model-based analysis of the TTEthernet startup and synchronization protocols.

We first use automated test-generation tools to drive high-coverage testing of prototype TTEthernet hardware, based on a state-machine model of the TTEthernet protocols. With almost no human guidance, this technique enables us to achieve MC/DC coverage of the startup protocol under valid fault scenarios.

We then focus on the TTEthernet clock-synchronization protocol. We develop correctness proofs of key properties of this protocol using the PVS interactive theorem prover [7]. As a result of this formalization, we have identified a suboptimal design choice in the clock-compression function defined in the TTEthernet draft standard [14]. We propose an alternative definition and, using model-checking tools, we show that the new function achieves better clock precision than the original.

These results demonstrate effective use of modeling and formal techniques in proof and test of a fault-tolerant network infrastructure relevant to avionics and other embedded systems.

Timed-Triggered Ethernet

TTEthernet [12,14] is a networking standard compatible with IEEE 802.3 switched Ethernet. It is designed to support data flows of mixed criticality on a single network. For traffic of the highest criticality, TTEthernet provides a timed-triggered communication service with strong guarantees of low jitter and bounded latency. This is achieved by maintaining a global time base across the network and by following a global communication schedule that prevents contention. TTEthernet also provides a rate-constrained communication service for traffic of intermediate criticality. For this traffic class, the worst-case communication latency can be computed offline, but it may be much higher than for timed-triggered traffic because rate-constrained messages from different sources may queue up in the network switches. Finally, traffic of the lowest criticality is transmitted using the standard, best-effort Ethernet approach, with no guarantees on transmission delays or message reception.

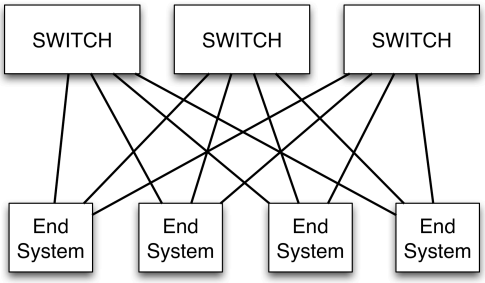

Figure 1. Example TTEthernet Network

Network Topology

A TTEthernet network consists of end systems and switches as depicted in Figure 1. The end systems are connected to switches by bidirectional communication links. Switches may be connected to each other in multi-hop network configurations. For fault tolerance, the network must be organized in disjoint redundant communication channels. Each channel consists of one or more switches that connect the end systems. Distinct switches must belong to

distinct channels so that a switch failure impacts only one channel.

Fault Models

TTEthernet can be configured for two different levels of fault tolerance. In a *single-failure* configuration, the network can tolerate the failure of a single component, which may be either a switch or an end system. In a *dual-failure* configuration, the network can tolerate two component failures. The faulty devices may be two switches, two end systems, or one switch and one end system.

In both configurations, the switches are assumed to have an *inconsistent-omission* failure mode. In the worst case, a faulty switch may drop or fail to receive an arbitrary number of messages on one or several of its ports, but it may not produce invalid messages. The failures may be asymmetric: some devices connected to a faulty switch may receive data while others do not.

The fault model for end systems depends on the configuration. In a single-failure configuration, a faulty end system may be Byzantine, that is, it may fail in an arbitrary manner. Under this assumption, the failure of an end system may have asymmetric manifestation or cause a "*babbling*" behavior. In a dual-failure configuration, the behavior of faulty end system is more restricted. It is assumed to be inconsistent omission.

TTEthernet Protocols

A major goal of TTEthernet is to ensure that all nodes in a network establish and maintain the common time base that is necessary for timed-triggered communication. During normal operation, all nodes must be closely synchronized and follow a common communication schedule that is computed offline. The common time base is a prerequisite to ensuring that timed-triggered traffic is deterministic and to providing guarantees of low jitter and fixed latency. Synchronization must be established and maintained despite the possible failure of switches and end systems.

To achieve these goals, TTEthernet includes a *startup protocol* that establishes synchronization after power up or restarts, a *clock-synchronization protocol* that maintains synchronization by periodically correcting possible clock drifts, and a *clique-detection and resolution service* to recover from network-wide transient upsets. In all these protocols, each device is assigned one of the following roles:

- Synchronization Masters (SM)
- Compression Masters (CMs)
- Synchronization Clients (SC)

SMs are responsible for starting up the network and for maintaining the synchronized time base. They initiate the startup protocol and, once the network is synchronized, they periodically trigger clock synchronization. All protocols start by the transmission of special Ethernet messages called protocol control frames (PCF) from one or more SMs to the CMs. The compression masters receive PCFs from the SMs. They filter, combine, and relay these PCFs to all nodes in the network. SCs have a passive role during startup and clock synchronization. They listen for communication and synchronize with the rest of the network on reception of PCFs that pass protocol-specific validity checks.

In typical networks, the SMs are end systems and the CMs are switches, although the standard allow other configurations [14]. In any case, the fault assumptions are as described previously for end systems and switches. In a single-failure configuration, the protocols are designed to tolerate either the Byzantine failure of an SM or the inconsistent-omission failure of a CM. In a dual-failure configuration, the protocols can tolerate the inconsistent-omission failure of at most two components. There are no significant assumptions on the failure of SCs since they are passive devices. In a multi-hop topology, the protocol still requires enough non-faulty components to ensure that PCFs can be routed through the network (i.e., that a sufficient number of independent channels is operational).

Existing TTEthernet Formalizations

Formal methods have been an integral part in the design of TTEthernet. In particular, the startup protocol was developed using SRI International's Symbolic Analysis Laboratory (SAL). SAL is a toolset for the analysis of state-transition systems

using model checking [6]. The heart of SAL is a language for specifying concurrent systems in a compositional way. SAL specifications can be analyzed using several model-checkers for finite and infinite-state systems.

Following the approach pioneered in [13], the startup protocol was developed and validated using the SAL tool chain. Formalizing the protocol definition in a form suitable for analysis using SAL made it possible to detect and address behavioral ambiguities early in the life cycle. It also resulted in the detection and removal of edge-case scenarios [12]. Early modeling, combined with feedback in the form of simulation and model-checking counterexamples, enabled the design team to develop an early intuition about the complex interactions between protocol components. This intuition was invaluable when the first protocol hardware implementations were debugged in the development laboratory. Other TTEthernet components, including several aspects of the clock-synchronization protocol, have also been formalized and verified using SAL [10,11].

This paper builds on these existing formalizations. First, we examine the use of SAL models to generate system-level test vectors for a representative TTEthernet network. We then report on formal verification of TTEthernet's *compression function*, a critical building block in the clock-synchronization service.

Model-Based Test Generation

TTEthernet prototype hardware has been subjected to traditional verification in the form of testing and simulation. The switches and end systems were treated as separate entities. Each component was individually verified using directed requirement-driven test campaigns, together with random testing based on System Verilog. These verification activities did not target the integrated system behavior of all TTEthernet components.

To explore the integrated system behavior, a network integration laboratory (NIL) was developed. It includes a test bed of more than 25 end systems and 17 switches, instrumented for fault injection. The NIL-based testing emphasized high-level system properties and did not target protocol branch coverage.

High-coverage testing of the TTEthernet startup protocol was seen as a necessary complement to the existing test results. First, we wanted to mitigate some of the risks associated with separate testing of switches and end systems, which may miss subtle interactions among the distributed components. Second, we wanted to validate the protocol soundness by presenting evidence that, under the core fault hypotheses, all the protocol logic is required and that no extraneous logic is present. We now describe the model-based method used for achieving high-coverage of TTEthernet startup.

SAL Model

We used a SAL model of the TTEthernet startup protocol that builds upon the model presented in [12]. This original SAL specification was modified and extended to a larger network that comprises six SMs and six CMs arranged as shown in Figure 2. Three redundant channels connect two sets of three synchronization masters, and each channel consists of two compression masters.

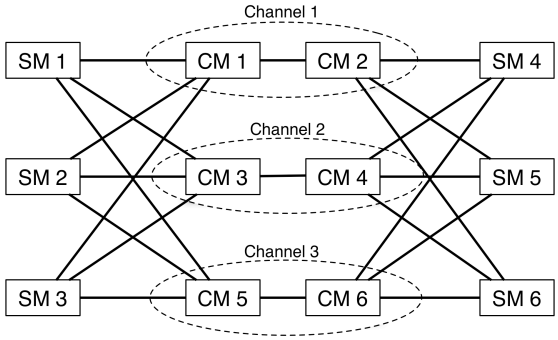

Figure 2. Test Network Configuration

In SAL, the SMs and CMs are modeled as finite state machines that encode the protocol states and actions defined in the TTEthernet standard. Timeouts and other timing constraints are modeled using a discrete time abstraction: time is represented by a finite interval. A small fragment of the SAL definition of a synchronization master is shown in Figure 3. The specification defines a protocol transition, from a state called SM_INTEGRATE to the state called SM_WAIT_4_CYCLE_START_CS. This transition is taken when the SM receives a cold-

start acknowledgement frame in state SM_INTEGRATE. Details on the interpretation of these states and messages are given in the TTEthernet standard [14]. The complete SAL specification is available on NASA's DASH link website https://c3.nasa.gov/dashlink/resources/593/

```
  SM_state = SM_INTEGRATE AND
  cs_ignore_states_ca AND
  NOT sm_sleep_timeout
-->
  trap_2' = TRUE;
  SM_state' = SM_WAIT_4_CYCLE_CS;
  SM_local_timer' = sm_ca_offset_timeout;
  SM_local_integration_cycle' = 0;
  SM_local_sync_membership'  = empty_membership
  SM_local_async_membership' = empty_membership;
  message_out' = [[ch: TYPE_CM_ids] empty_message];
  ...
```

Figure 3: SAL Model Fragment

As shown in the figure, state transitions are specified in SAL using a guarded command notation of the form

guard --> variable updates

The guard is a Boolean condition that defines when the transition is enabled (i.e., when it may be taken), and the variable updates define the effect of this transition of the system's state.

In addition to encoding the protocol rules, the SAL model is equipped with Boolean flags that correspond to the fault-injection capabilities of the hardware test bed. For example, Boolean flag sm_sleep_timeout in Figure 3 indicates that the SM has been forced into a sleep state by the testing environment. When this flag is true, the SAL model does not respond to any stimulus from the network; the transition is disabled. In the SAL model, variable sm_sleep_timeout is unconstrained. It can be set non-deterministically to true or false at every protocol step.

The variable trap_2 also plays a special role. It is initially false; it is set to true when the transition in Figure 3 is taken; and it is left unchanged by all other transitions in the SAL model. In other words, trap_2 indicates whether the transition from SM_INTEGRATE to SM_WAIT_4_CYCLE_CS has ever been executed.

The full SAL model is the synchronous composition of modules describing the synchronization and compression masters, together with a module that models the connections between SMs and CMs. At each step, the connection module selects the messages to be delivered. The network topology is encoded into connectivity constraints. For example, there is no direct link in Figure 2 between CM1 and CM4, so the connection module never delivers to CM4 any message sent by CM1 and vice versa. Other constraints in the connection module encode the fault model. For example, if CM1 is faulty then it may fail to transmit to some of its neighbors. This is encoded in the SAL model by non-deterministically selecting a subset of CM1's neighbors to which the messages from CM1 are delivered. This set of neighbors can change arbitrarily at each protocol step.

Test-Generation Tools

Analysis of the TTEthernet startup model relies on SAL's automated test-generation tool called sal-atg, which is described in details in [4]. This tool attempts to find an input sequence for a SAL model that will cause the system under test to exhibit behaviors of interest, the *test goals*. The tool generates test sequences from a SAL system that has been augmented with *trap variables* that describe the test goals. These variables are initially false and are set true when a specific test goal has been satisfied. Variable trap_2 in Figure 3 is an example of such trap variables. If trap_2 is given as one of the test goals to sal-atg, the tool will search for a protocol execution that sets trap_2 true, that is, for an execution in which the transition of Figure 3 is taken (at least once).

Sal-atg is highly flexible, and it can use a combination of model-checking techniques to produce sequences that satisfy the test goals. In our analysis of TTEthernet startup, we relied exclusively on the bounded model checking capabilities of sal-atg. Bounded model checking was introduced in [1]. It is based on converting the search for execution traces that satisfies certain properties (in our case, meet the test goals) into an equivalent Boolean satisfiability problem.

Bounded-model checking has become a very efficient analysis technique since the emergence of powerful Boolean SAT solver. In our analysis of TTEthernet, we used the state-of-the-art solver plingeling [2] as a backend solver to the sal-atg tool. Plingeling is a multi-threaded SAT solver that can solve very large problems containing millions of Boolean variables and clauses.

Model Validation

The SAL model we used in this work extended the original model from [12] in several ways. It included new features of the TTEthernet startup protocol that were not present in the original model, and it considered a larger network configuration. Because the differences were substantial, we had first to validate our revised model. We first checked that a critical property of TTEthernet startup protocol that holds in the original model was still satisfied after the modifications. This property is an upper bound on the time it takes for the network to initially synchronize. By using the SAL model-checking tools, we showed that this property was still satisfied in the extended model. This result confirmed our expectation that the worst-case startup time is less than 60 protocol steps, and showed that the extended model behaved consistently with the original model.

For additional validation, we explored the SAL model to show that its executions were consistent with the designers' understanding of the protocol. For this purpose, we examined several interesting scenarios that the TTEthernet designers knew could be observed during startup. Using sal-atg, we showed that these scenarios could also happen in the SAL model. This success increased our confidence in the correctness of the model. The exact scenarios investigated and the results from sal-atg can be found at https://c3.nasa.gov/dashlink/resources/593/.

Coverage-Based Test Generation

The remainder of the analysis aimed to generate high-coverage test vectors that exercise all the startup protocol logic. More specifically, our goal was to achieve MC/DC coverage of the startup protocol, under fault scenarios consistent with TTEthernet's fault model. We allowed for as many as two faulty CMs to be present within a three-channel system. We considered both the single-failure and dual-failure hypotheses.

In each test scenario, sal-atg constructs an execution sequence in which all non-deterministic choices present in the SAL model are resolved. Sal-atg decides when each device is powered on or off (by setting variables such as sm_sleep_timeout to true or false) and which messages from faulty components are received or dropped. To reproduce these test cases on actual hardware, we had to restrict the fault model to permanent communication failures. That is, connection failures were held consistent throughout the entire test scenario. This decision simplified the execution of tests on the hardware and did not affect the coverage results. On the other hand, the test generation could dynamically power on the SMs or put them to sleep at any time, and could delay power on of the non-faulty CMs. This level of control is aligned with the capabilities of the TTEthernet hardware validation test bed.

```
  A and (B or C)
-->
  trap_3a' = B;
  trap_3b' = C;
  more variable updates
```

Figure 4. Trap Variables for MC/DC Coverage

The first step of MC/DC test coverage was to instrument the SAL model. We added trap variables to every transition of the SM and CM state machines (as shown in Figure 3). Furthermore, when the guard of a transition involved a logical OR, additional trap variables were introduced to record the independent impact of each condition in the guard, as illustrated in Figure 4. The two trap variables shown in the figure capture the two possible ways in which the guard can be true: either both A and B are true (trap_3a), or A and C are true (trap_3b). To achieve MC/DC coverage, we aim to generate tests that independently set both trap variables to true.

Once the SAL model was instrumented, we ran sal-atg with different test goals and for two variants of the model that encoded the two possible fault hypotheses (either single or dual failures). The different test goals were selected to focus on MC/DC coverage of either the SM state machine or the CM state machine. We also investigated MC/DC

coverage of another SAL module that models an optional priority scheme defined by the TTEthernet standard. In all these different experiments, using sal-atg was straightforward. We just gave the trap variables of interest as test goals.

Test coverage can be considered at the system level and from the perspective of a single SM or CM. At the system level, MC/CD coverage requires us to exercise each protocol action in one of the network's SM or CM; different actions may be covered by different device. In practice, it is more useful to achieve high coverage from the perspective of a single device, as this simplifies hardware instrumentation and testing. We have performed two series of test-generation experiments, aiming to achieve both system-level and component-level coverage. The results presented next correspond to component-level coverage.

Results

The performance of sal-atg in conjunction with the plingeling SAT solver was impressive. For all the coverage models, the tests generated by sal-atg achieved full coverage of all reachable state transitions. For each fault hypotheses, the SM coverage runs completed in approximately two days. Test generation runs for the CM coverage and for the priority module completed is about a day or less. In all cases, the SAT solver runtime dominates.

We encountered memory problems when we tried to run sal-atg with a set of 54 trap variables for MC/DC coverage of the SM state machine. We solved this problem by splitting the set into five smaller subsets and running sal-atg on each subset independently.

Table 1 summarizes the test-generation results and runtimes for coverage of the CM and SM state machines, and for the two fault hypotheses considered by TTEthernet. The second column shows the number of test goals discharged by sal-atg compared to the number of test goals given as input. For example, in the CM/single fault run, sal-atg generated tests that covered 24 out of the 31 test goals given as input.[1]

Table 1. Test-Generation Results

Experiment	Test goals discharged	Run time
SM/single fault	49/54	74 h 5 min
SM/two faults	50/54	74 h 40 min
CM/single fault	24/31	17 h 7 min
CM/two faults	17/19	28 h 20 min

In all cases, sal-atg generates test sequences that discharge almost all the test goals given. Further inspection showed that the test goals missed by sal-atg are not reachable at all in the network configuration that we studied. Hence, sal-atg discharged all the test goals that could be reached in this model.

The runtimes are of the orders of a day or two, which is quite good considering the size of the SAL models, and the complexity of the TTEthernet startup protocol. No doubt achieving the same coverage with hand-generated tests would take a lot more time and effort. During the test-generation effort, we found that using a state-of-the-art SAT solver such as plingeling, which can take advantage of multicore machines, had a significant impact on runtime. Using plingeling instead of the default SAT solver that comes with sal-atg reduced the runtimes by a factor of four to five. All experiments were run on a standard, multicore desktop computer running Linux 2.6.

Analysis of TTEthernet's Compression Function

The startup protocol brings the network from an initial unsynchronized state to the synchronized state necessary for timed-triggered operation. To maintain

[1] In TTEthernet, the CM state machine is more complex in the single-fault hypotheses than in the two-fault model. This explains why the CM/single fault run has more test goals than the CM/two faults run.

synchronization, the network must periodically run a *clock-synchronization protocol* to correct clock drift. This protocol is intended to maintain a given network-wide clock precision, even in the presence of faulty nodes. In addition to the test-generation work presented previously, we have developed a formal model of the clock synchronization protocol, with the aim to verify its correctness.

Clock Synchronization Overview

In a TTEthernet network, all timed-triggered communication follows a global, periodic schedule. This communication schedule consists of a *cluster cycle* divided in a finite number of *integration cycles* of equal duration. The TTEthernet synchronization protocol is executed periodically, at the beginning of each integration cycle.

The SMs trigger the protocol execution by sending their local clock values to the CMs, within so-called *integration PCFs*. Each CM collects the integration frames it receives and records the reception times, as measured by the CM's local clock. Integration frames are labeled with the integration cycle in which they originated. A CM groups the integration frames based on the integration cycle they contain and on the time when they were received (see [14] for details). For each group of integration frames, the CM computes a fault-tolerant average of the received values by applying the TTEthernet *compression function*. The CM applies validity checks to verify that the compression value comes from a sufficient number of SMs and is not too far from its local clock. If the compression value passes these validity checks, the CM uses it to correct its local clock. In addition, the CM broadcasts the compression results to the network. At this point, both SMs and SCs receive compression values from one or more CMs. They apply local validity checks to filter out bad compression values then they use another averaging function to compute a correction for their local clock.

Formalization and Proofs

Establishing the correctness of a fault-tolerant clock-synchronization protocol is a difficult and error-prone exercise, which can be helped by the use of formal verification tools such as theorem provers. Such tools enable users to formalize protocol model and develop detailed and rigorous proofs that the protocols work properly. Several clock-synchronization protocols from the literature have been verified using the PVS theorem prover, and its predecessor EHDM [5,8,9]. In some cases, the formalization uncovered subtle imprecision and flaws in published hand proofs.

In the case of TTEthernet, we have developed SAL models of some aspects of the clock synchronization protocol, and established correctness properties using bounded model checking [10,11]. However, the SAL models developed for this purpose abstracted away some of the protocol mechanisms, and the formalization considered only a limited set of small instances of TTEthernet (with as many as six SMs and two CMs). We wanted to extend these results to the general case of networks with an arbitrary number of SMs and CMs. As part of this effort, we have focused on the TTEthernet compression function, which is crucial to the correctness of the clock-synchronization protocol. We now summarize the main results of this formalization. The complete PVS developments are available on NASA's DASHLink server at https://c3.nasa.gov/dashlink/resources/601/.

```
n: VAR nat

cvector(n): TYPE+ = [below(n) -> clock_time]

m: VAR posnat

compress(m)(v: cvector(m)): clock_time =
  COND
     m = 1  ->  v(0),
     m = 2  ->  avg(v(0), v(1)),
     m = 3  ->  v(1)
     m = 4  ->  avg(v(1), v(2)),
     m = 5  ->  v(2),
     ELSE  -> avg(v(K), v(m-K-1)
  ENDCOND
```

Figure 5. The Compression Function in PVS

PVS Formalization

The core of the TTEthernet clock correction protocol is function compress shown in Figure 5. This function takes a finite vector v of clock values as input and computes an average of v's components. The vector v must be sorted in increasing order. The

actual averaging function applied depends on the size of the vector. For example, if v contains three, four, of five elements, then compress returns the median of these elements. The parameter K shown in Figure 5 is the maximal number of faulty SMs to tolerate (i.e., K is one in a single-failure configuration, and K is two in a dual-failure configuration), and function avg computes the average of two numbers.

A CM applies this compress function to a set of integration PCFs it receives from SMs. This requires first sorting the PCFs in increasing order of reception time, and computing clock differences. The details of the full procedure are not show in the figure but are available in the full PVS specification.

```
comp_correction(B): clock_time = compress(card(B))(clock_diffs(B)

cm_compressed_pit(B): clock_time =
  start_perm(B) + max_observation_window +
    calculation_overhead + comp_correction(B)

similar_synchronized_convergence: PROPOSITION
        similar(C1, C2, I, eps)
  AND synchronized(C1, I, precision)
  AND synchronized(C2, I, precision)
  AND card(C1) = m1 AND card(C2) = m2 AND card(I) = n
  AND n >= 2 * K + 1  AND  n >= m1 - K  AND  n >= m2 - K
  AND m1 /= 5 AND m2 /= 5
 IMPLIES
     abs(cm_compressed_pit(C1) - cm_compressed_pit(C2))
       <= precision/2 + 1  + eps
```

Figure 6. Main Property of the Compression Function

The key property that we have proved using PVS is shown in Figure 6. In this PVS fragment, C1 and C2 denote two sets of integration frames received by two distinct compression masters, and I denotes the set of non-faulty synchronization masters. The constant precision is the clock precision that is assumed to hold before the clock synchronization protocol is executed, and constant eps denotes the imprecision in communication latency. Essentially, the convergence property of Figure 6 expresses that applying the compression function reduces the worst-case distance between the local clocks of two non-faulty CMs. Before clock correction, two CM clocks may differ by as much as the precision. After clock correction the difference is no more than half the precision plus a small error term. This property explains why the clock compression function compensates for clock drift. The property holds under various constraints on the number of good SMs, and the cardinalities of C1 and C2, in relation to K, the maximal number of faulty SMs.

An Issue With the Compression Function

A surprising result of the PVS developments summarized previously is that the key convergence property in Figure 6 does not hold when C1 or C2 contain exactly five PCFs. This points to an oversight in the definition of the compress function. When applied to a vector of five elements, compress returns the median. But, in a scenario with four good SMs and one Byzantine-faulty SM, the latter can essentially determine the median. If the four good values are $c_0 \le c_1 \le c_2 \le c_3$, then the Byzantine SM can force the median to be c_1 by producing a faulty value smaller than c_0, or it can force the median to be c_2 by producing a value larger than c_3. Thus, an asymmetric fault can cause some CMs to synchronize with c_1 and others to synchronize with c_2. In the worst case, the difference between these two values is precision + eps, so the clock skew may increase. The same result is possible without Byzantine failures, if two out of six SMs have inconsistent-omission failures.

These failure scenarios show that the CMs may not be synchronized as closely as one would expect. However, this reduced precision does not lead to a complete loss of network synchronization. The SMs and SCs apply another averaging function to the compressed clock values they receive from CMs. These additional mechanisms do not allow the errors to accumulate, since, as shown in [10], the SMs are synchronized with each other. The whole network will then remain synchronized, even in the pathological cases identified previously, but with some degradation in the clock precision achieved. We have not investigated this issue very deeply since there is a simple fix to the compression function.

A Revised Compression Function

We propose the following revised definition for the compression function:

```
compress(m)(v: cvector(m)): clock_time =
  COND
     m = 1  ->  v(0),
     m = 2  ->  avg(v(0), v(1)),
     m = 3  ->  v(1)
     m = 4  ->  avg(v(1), v(2)),
     m = 5  ->  avg(v(1), v(3)),
     ELSE  -> avg(v(K), v(m-K-1)
  ENDCOND
```

With this new definition, the compression of a vector v of five clock readings is the average of the second and fourth value, instead of the median of the five values. With this revision, one can show that the convergence property (Figure 6) now holds even when the input sets C1 and C2 contain five integration frames.

Validation of the Revised Function

By using another tool in the SAL system, we can now compare the two definitions of the clock compression function. We build a simplified SAL model of the synchronization protocol, and we compare the worst-case clock drift between different network components. The SAL model generalizes a previous formalization presented in [2], which focused on bounding the clock drift between SMs.

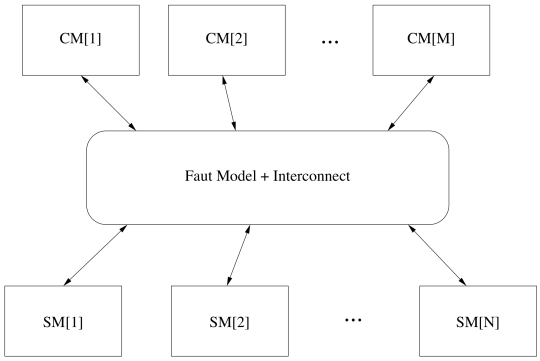

Figure 7. SAL Model for Analysis of the Compression Function

The SAL model we develop is structured as shown in Figure 7. It consists of independent processes that represent the CMs and SMs, and an interconnect module that specifies how the output from each process is received by other processes. Faults are modeled in the interconnect module. If a source process is non-faulty, then its output is received unchanged by all recipients. Otherwise, the recipients may see different input depending on the source's fault. For example, if the source has an inconsistent-omission fault, then some recipients receive the data as sent while others receive nothing. This SAL model is described in detail in [3] and is available at https://c3.nasa.gov/dashlink/resources/601/.

Unlike the SAL models discussed previously, the model we used for analyzing the compression function is not finite state, since the clock of each component is represented by a real-value variable. Analysis of such SAL models can still be performed using SAL's bounded model checker for infinite-state systems called sal-inf-bmc. Using this model checker, we can prove the clock-precision bounds shown in Figures 8 and 9.

```
sm_clock_distance: LEMMA
  TTE |- G(FORALL (i, j: SM_ID): sm_clock[i] - sm_clock[j] <= 2 * max_drift);

sm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i, j: SM_ID): sm_clock[i] - sm_clock[j] <  2 * max_drift);

cm_clock_distance: LEMMA
  TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 4 * max_drift);

cm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <  4 * max_drift);

sm_cm_clock_distance: LEMMA
  TTE |- G(FORALL (i: SM_ID, j: CM_ID):
                sm_clock[i] - cm_clock[j] <= 3 * max_drift AND
                cm_clock[j] - sm_clock[i] <= 3 * max_drift);

sm_cm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i: SM_ID, j: CM_ID):
                sm_clock[i] - cm_clock[j] <  3 * max_drift AND
                cm_clock[j] - sm_clock[i] <  3 * max_drift);
```

Figure 8. Clock Precision for the Original Compression Function

The results of Figure 8 correspond to a baseline SAL model that uses the original compression function. This model includes five synchronization masters and two compression masters, with the assumption that one synchronization master is Byzantine faulty. Figure 8 shows six properties, organized in three groups of two lemmas. The first lemma in each pair was proved with sal-inf-bmc. It establishes an upper bound on the difference between the clocks of two components. The second lemma in each pair is false. Counterexamples can be found using sal-inf-bmc, which shows that the bound given by the first lemma is precise. All bounds are

expressed as multiples of max_drift, which denotes the maximal drift that a clock can experience in one integration cycle. For example, the difference between the clocks of two compression masters can be equal to four times the maximal drift.

Figure 9 shows the same results for a SAL model that uses the revised compression function. Again, this SAL model includes five SMs and two CMs, and assumes that one of the SMs is Byzantine. As can be seen in Figure 9, the clock precision is improved. In particular, the maximal difference between the clocks of two CMs is now three times max_drift instead of four times max_drift.

```
sm_clock_distance: LEMMA
  TTE |- G(FORALL (i, j: SM_ID): sm_clock[i] - sm_clock[j] <= 2 * max_drift);

sm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i, j: SM_ID): sm_clock[i] - sm_clock[j] <  2 * max_drift);

cm_clock_distance: LEMMA
  TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 3 * max_drift);

cm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <  3 * max_drift);

sm_cm_clock_distance: LEMMA
  TTE |- G(FORALL (i: SM_ID, j: CM_ID):
                sm_clock[i] - cm_clock[j] <= 5/2 * max_drift AND
                cm_clock[j] - sm_clock[i] <= 5/2 * max_drift);

sm_cm_clock_distance_strict: LEMMA
  TTE |- G(FORALL (i: SM_ID, j: CM_ID):
                sm_clock[i] - cm_clock[j] <  5/2 * max_drift AND
                cm_clock[j] - sm_clock[i] <  5/2 * max_drift);
```

Figure 9. Clock Precision for the Revised Compression Function

These results were obtained for a simplified model of the TTEthernet synchronization protocol. However, they establish convincingly that the revised compression function is better than the original, by improving the synchronization quality. We have reported the results of our analysis to the TTEthernet designers, and the revised compression function has now been fully implemented in the published TTEthernet standard SAE AS6802.

Conclusion

Formal method tools based on model-checking, SAT solving, and other technology can be effective in model-based design and analysis of industrial protocols such as TTEthernet. We have demonstrated the power of these tools during protocol design, testing, and verification. Key results include the ability of modern model-checking technology to generate high-coverage test vectors for a complex real-time protocol. This success is due in large part to the ability of bounded-model checkers to leverage the impressive performance of recent SAT solvers. We have also demonstrated how a deeper protocol analysis using theorem proving led to the discovery of a suboptimal design in TTEthernet's compression function. The fix we proposed was validated using another form of bounded model checking that enables analysis of infinite-state real-time systems, and it has been incorporated into the released TTEthernet standard.

In future work, we are planning to use the SAL-generated test vectors to test the hardware implementation of TTEthernet. The test-generation experiments have also identified improvements to the sal-atg tool, such as the ability to generate tests from a specified set of initial states, rather than form the fixed initial conditions specified in the SAL model. We are also planning to complete a full formal verification of the TTEthernet protocol suite, to complete the current verification results that have each focused on a different aspect of TTEthernet.

Acknowledgements

The first three authors were supported by NASA contract NNL10AB32T. The first author was also partially supported by NSF Grant CSR-0917398. The content is solely the responsibility of the authors and does not necessarily represent the official views of NASA or NSF.

References

[1] Biere, Armin, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu, 1999, Symbolic Model Checking without BDDs, Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS 1579, Springer-Verlag, pp. 193–207.

[2] Biere, Armin, 2011, Lingeling and Friends at the SAT Competition 2011, Technical Report 11/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria.

[3] Dutertre, Bruno, Natarajan Shankar, Sam Owre, 2012, Integrated Formal Analysis of Timed-Triggered Ethernet, NASA Contractor Report, CR-2012-217554, NASA.

[4] Hamon, Grégoire, Leonardo de Moura, John Rushby, 2004, Generating Efficient Test Sets with a Model Checker, 2nd International Conference on Software Engineering and Formal Methods, IEEE Computer Society, pp. 261–270.

[5] Miner, Paul, 1993, Verification of Fault-Tolerant Clock Synchronization Services, Technical Paper 3349, NASA Langley Research Center.

[6] de Moura, Leonardo, et al., 2004, SAL 2, Computer-Aided Verification (CAV 2004), LNCS 3114, Springer-Verlag, pp. 496–500.

[7] Owre, Sam, John Rushby, Natarajan Shankar, Friedrich von Henke, 1995, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, IEEE Transactions on Software Engineering, vol. 21, number 2, pp. 107–125.

[8] Rushby, John, Friedrich von Henke, 1989, Formal Verification of the Interactive Convergence Clock Synchronization Algorithm, Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International.

[9] Shankar, Natarajan, 1991, Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm, Technical Report CR-4386, NASA.

[10] Steiner, Wilfried, Bruno Dutertre, 2011, Automated Formal Verification of the TTEthernet Synchronization Quality, NASA Formal Methods Conference (NFM 2011), LNCS 6617, Springer-Verlag, pp. 373–390.

[11] Steiner, Wilfried, Bruno Dutertre, 2010, SMT-Based Formal Verification of a TTEthernet Synchronization Function, Formal Methods for Industrial Critical Systems (FMICS 2010), LNCS 6371, Springer-Verlag, pp. 146–164.

[12] Steiner, Wilfried, 2009, TTEthernet Executable Formal Specification, CoMMiCS Project Deliverable.

[13] Steiner, Wilfried, John Rushby, Maria Sorea, Holger Pfeifer, 2004, Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation, Proc. 2004 International Conference on Dependable Systems and Networks (DSN'04), IEEE, pp. 189–198.

[14] Timed-Triggered Ethernet. SAE Aerospace Standard AS 6802, v1.1.2, 2011. Draft.