NASA/CR–2012-217554

Integrated Formal Analysis of Timed-Triggered Ethernet

Bruno Dutertre, Natarajan Shankar, and Sam Owre
SRI International, Menlo Park, California

March 2012

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2012-217554

Integrated Formal Analysis of Timed-Triggered Ethernet

Bruno Dutertre, Nstarajan Shankar, and Sam Owre
SRI International, Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

March 2012

Available from:

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

Abstract

We present new results related to the verification of the Timed-Triggered Ethernet (TTE) clock synchronization protocol. This work extends previous verification of TTE based on model checking. We identify a suboptimal design choice in a compression function used in clock synchronization, and propose an improvement. We compare the original design and the improved definition using the SAL model checker.

Contents

1 Introduction

TTEthernet extends traditional IEEE 802.3 Ethernet networks with new services and protocols to support reliable, deterministic communication for real-time applications [1, 18]. Several TTEthernet components have been modeled and analyzed using the SAL model checker. This has resulted in a collection of disparate models, each built for a specific protocol and for a specific type of analysis. Finite-state representations of the startup protocol were used for model-based test generation [7, 17]. Other infinite models were developed for verifying timing properties of TTEthernet services such as the *permanence* and *compression* functions [19], or compute the precision of the clock-synchronization service [20]. Thus, several important pieces of TTEthernet have been the subject of extensive analysis using formal-method tools. Each of these previous analyses relied on model checking technology and was applied to finite instances of TTEthernet that consisted of a finite number of components and a fixed topology. How can we extend these piecewise verifications into a coherent and complete argument about the correctness and reliability of the whole TTEthernet concept?

We do not yet have a full end-to-end correctness argument for TTEthernet, but we present recent work on formalizing and verifying part of the clock-synchronization protocol of TTEthernet using the PVS interactive theorem prover. We show that the current definition of the TTEthernet compression function is suboptimal and we propose a simple fix. We present a revised model of the clock synchronization protocol in SAL to examine the impact of the revised compression function on clock precision. We also describe the current status of SRI's Evidential Tool Bus (ETB), a framework intended to facilitate the integration of several modeling and verification technology. Our long-term goal is to complete the TTEthernet modeling and verification within the ETB.

In the remainder of this report, we first give an overview of TTEthernet in Section 2. We then present the existing verification results obtained using the SAL model checker, and survey existing formalization of fault-tolerant clock synchronization protocols that use the PVS theorem prover. We discuss the issues encountered in our attempts to combine these past results, and we propose a variant formalization approach intended to facilitate combined analysis using model checker such as SAL and interactive theorem provers such as PVS. Section 4 summarizes the current status of a PVS formalization of the TTEthernet clock synchronization protocol. Section 5 presents a (simplified) SAL model of the protocol that can be used to estimate maximal clock skews between network components. This new model revises and extends previous work on the quality of the clock-synchronization service presented in [20]. We then describe the ETB and our plans for using it as an integration framework in combined PVS/SAL verification of TTEthernet. The full SAL models and PVS theories discussed in this report can be found at `http://www.csl.sri.com/users/bruno/vvfcs.html`.

2 TTEthernet

TTEthernet [23] is a communication infrastructure that enables the use of Ethernet in real-time, distributed systems. TTEthernet is compatible with IEEE 802.3 switched Ethernet standards, and is designed to support dataflows 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 transmission latency can be computed offline but it may be much higher than for timed-triggered messages 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

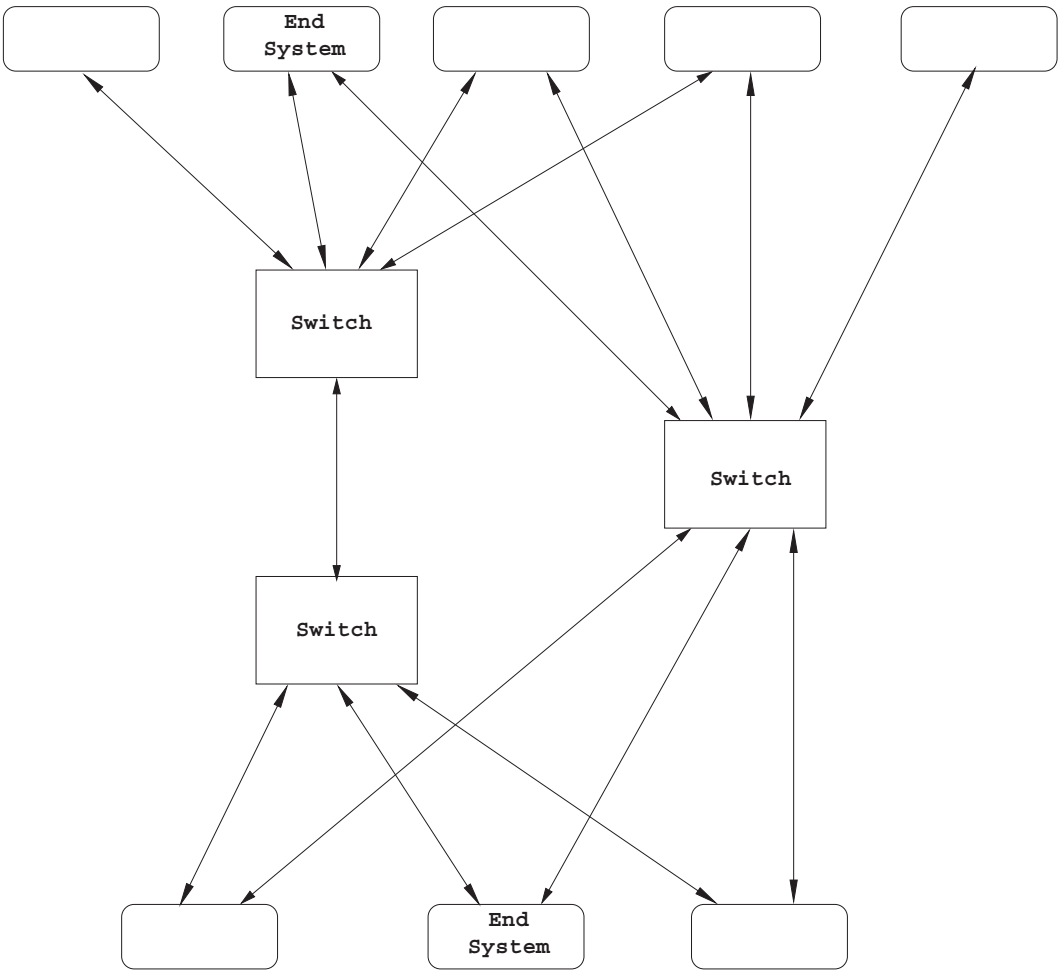

Figure 1. Example TTEthernet Network

message reception.

2.1 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 multihop network configurations. For fault tolerance, the network may 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.

2.2 Fault Tolerance

TTEthernet networks can be configured for different levels of fault tolerance. In a *single-failure* configuration, the network can tolerate the failure of one component, which may be either an end system or a switch. 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, 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 failure 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. Failure of an end system may then have an asymmetric manifestation or cause a "babbling idiot" behavior. In a dual-failure configuration, the behavior of faulty end systems is assumed to be *inconsistent omission*.

2.3 Clock Synchronization Overview

The major goal of TTEthernet is to ensure that all nodes establish and maintain the common time base that is necessary for timed-triggered operation. During normal operation, all nodes must be closely synchronized and follow a global 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 latency. For safety-critical applications, synchronization must be maintained despite the possible failures of switches or end systems.

To achieve these goals, TTEthernet includes a *startup protocol* that establishes synchronization after power-up or restart, a *clock synchronization* protocol that maintains synchronization by correcting possible clock drifts, and a *clique detection and resolution* service to recover from network-wide transient upsets.

All these protocols are described in detail in the TTEthernet standard [23]. We focus here on the clock-synchronization protocol. In this protocol, each network device is assigned one of the following roles:

- *Synchronization Master (SM).* Synchronization masters trigger execution of the clock-synchronization protocol by periodically broadcasting their local clock value within special Ethernet messages called *process control frames* (PCFs).

- *Compression Master (CM).* A compression master receives clock values from the synchronization masters and computes an average of the received values by applying a fault-tolerant *compression function*. The compression master uses the average to correct its own local clock. It also broadcasts the compression result to the network in a compression PCF.

- *Synchronization Clients (SC).* Synchronization clients are all nodes in the networks other than the SMs and CMs. A synchronization client has a passive role during clock synchronization. It waits for compression PCFs from the CMs and it computes a clock correction for its local clock by averaging the compression values it receives. The same operation is also performed by the SMs. After broadcasting their local clock value, the SMs wait for compression values from the CMs and use them for correcting their local clock.

In typical networks, the SMs are end systems and the CMs are switches, although this is not strictly required by the TTEthernet standard [23]. In any case, the fault assumptions for SMs and CMs are as described previously for the end systems and switches:

- In a single-failure configuration, the protocol is designed to tolerate either the Byzantine failure of a single SM, or the inconsistent-omission failure of a single CM.

- In a dual-failure configuration, the protocol can tolerate the inconsistent-omission failure of two components (either two SMs, or two CMs, or one SM and one CM).

There are no significant assumptions on the failure of SCs since the SCs are passive during clock synchronization. In a multihop topology, the protocol still requires enough nonfaulty components to ensure that

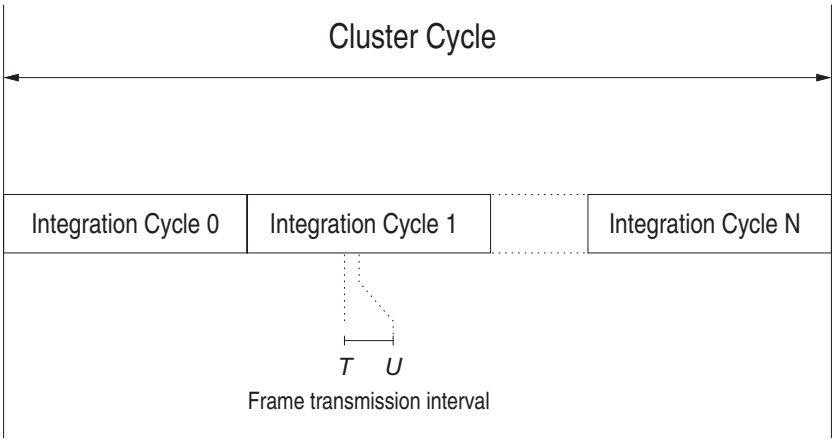

Figure 2. TTEthernet Schedule

messages can be routed through the network (i.e., that a sufficient number of independent channels are operational).

The goal of the clock-synchronization protocol is to maintain a given network-wide clock precision, even in the presence of faulty nodes. This ensures that all nodes agree on the current communication-schedule slot and thus that timed-triggered traffic is conflict free: two timed triggered frames never contend for access to a communication link.

More precisely, a global TTEthernet communication schedule is periodic. It consists of a *cluster cycle* divided in a finite number *integration cycles* of equal nominal duration, as depicted in Figure 2. The timed-triggered traffic consists of a number of message (or frames) that are routed from one source node to one or more destination nodes through the serial links and switches. The communication schedule defines the time when each frame should be transported across each link along its route. In other words, if a frame f must be transported across a link l, then the schedule assigns a transmission interval $[T_{f,l}, U_{f,l}]$ for that frame within the cluster cycle. The schedule ensures that transmission of other timed-triggered frames over link l does not overlap with interval $[T_{f,l}, U_{f,l}]$. More details on TTEthernet scheduling problem are presented in [16].

The clock-synchronization protocol is executed at the beginning of every integration cycle. By maintaining the clocks synchronized, the protocol ensures that all nodes agree on when each integration cycle starts and on when each timed-triggered frame should be transmitted. For example, if link l is between two nodes i and j, then node i dispatches frame f at a time t_i when its local clock $C_i(t_i)$ reads $T_{f,l}$. This time t_i must be close to the time t_j when node j's clock reaches $T_{f,l}$. More generally, the goal of the protocol is to ensure that the clocks C_i and C_j reach a schedule point T at approximately the same real times t_i and t_j:

$$\forall T : C_i(t_i) = T \ \wedge \ C_j(t_j) = T \ \Rightarrow \ |t_i - t_j| \leq \Delta$$

where Δ is the clock precision.

Our goals are to develop a rigorous proof that the clock synchronization protocol actually satisfies this property, for both the single-failure and dual-failure configurations, and to establish sufficient conditions on the protocol parameters— such as the number of SMs and CMs, and the length of the integration cycles— under which correct synchronization can be achieved.

3 Toward Combining Multiple Formalizations

Verification of fault-tolerant clock-synchronization protocols using formal-method techniques is not new. We summarize some of existing work in this domain, and we survey previous verifications of parts of TTEthernet. We then explain difficulties we encountered when trying to combine and reuse this existing work and apply it to the full TTEthernet clock-synchronization protocol.

3.1 Generic Clock-Synchronization Proofs

Several protocols from the literature have been verified using the PVS theorem prover and its predecessor, EHDM. Rushby and von Henke formalized and verified Lamport and Meliar-Smith's protocol using EHDM [13]. The formalization uncovered subtle imprecisions and flaws in Lamport and Meliar-Smith's original proof [8]. In subsequent work, Shankar [15] used EHDM to verify a generic Byzantine clock synchronization protocol due to Schneider [14]. Miner [11] later improved and extended Shankar's formalization, and instantiated the generic proof to an algorithm and hardware implementation related to the Welch-Lynch clock-synchronization protocol [9].

Similar proofs have been developed using other theorem provers such as Isabelle/HOL [2] or the Boyer-Moore theorem prover [24].

More recently, Miner et al. [10] developed a unified model for fault-tolerant algorithms, and showed that Byzantine clock synchronization can be seen as an instance of this generic scheme. The unified model was formalized and verified using PVS, and was applied to show correctness of SPIDER's clock-synchronization algorithm [22].

Most of these existing verifications have a similar formalization style, which is close to the traditional mathematical presentation found in the literature on clock-synchronization algorithms [8, 9, 14]. A clock is typically modeled a mathematical functions C that maps real time to clock time (i.e., $C(t)$ is the clock value of C at time t)[1] and assumptions on clock drifts are expressed as constraints on the increase rate of these functions. For each protocol participant, the synchronization is modeled as a sequence of clocks $C_0, C_1, \ldots$, each of them used for a finite time interval. The protocol rules define how C_{i+1} is constructed from preceding C_js and specify when each component should switch over from clock C_i to clock C_{i+1}. Correctness proofs require bounding the difference between the clock functions of distinct components, which typically requires induction on the sequence of clocks and time intervals.

3.2 Existing TTEthernet Formalizations

TTEthernet itself has been the focus of significant formal modeling and verification. TTEthernet's design has been guided by extensive analysis of variant protocols and algorithms, using state-machine models and model checking [21]. In particular, the startup protocol was designed and extensively tested using the SAL model checker. A fixed-size instance of the protocol was modeled as a composition of finite-state machines, and other modules were used to model faults. On such a finite model, bounded model checking showed that the startup protocol successfully brings all nodes from an unsynchronized to a synchronized state, even in the presence of two component failures. This analysis also determined the worst-case startup or restart time from different clique scenarios.

Using a different modeling approach, based on the SAL infinite-state bounded model checker, we have also verified correctness properties of two TTEthernet functions that are used by the synchronization protocol [19]. The *permanence function* is a simple method for controlling jitter in protocol frames. This mechanism makes it look as if all protocol frames have maximal latency but low jitter, that is, the variation

[1] The reverse convention is used too. A clock can be modeled as a function c from clock time to real time, with the interpretation that $c(T)$ is the real time at which clock c has value T.

in transmission delays appears very small to the frames' consumers. Low jitter is important for synchronization as it determines how accurately one node can estimate the clock of another node in the network. The permanence function is implemented by keeping track of the actual transit delay experienced by a frame, and artificially adding more delay at the destination node before delivering the frame to the higher levels in the protocol stack.

The SAL model used in this work builds upon the timeout and calendar automata framework described in [5, 6]. This model is inspired by the event calendar (or even lists) commonly used in discrete-event simulation, and has proved useful for analyzing several types of timed systems with SAL [3, 4]. In the TTEthernet context, the calendar and timeout automata model enables us to capture clock drift, delays in message transmission, and protocol mechanisms such as timers. The resulting SAL models are not finite since some of the model variables are real-valued (e.g., they encode delays and timeouts), however such systems are still amenable to verification using `sal-inf-bmc`, the SAL bounded model checker for infinite state systems. This bounded model checker relies on modern SMT solvers such as Yices that can decide the satisfiability of logical formulas involving arithmetic, Booleans, and other useful theories. In this framework, correctness properties can be established by k-induction. Proofs are not always fully automatic and may require human guidance in the form of auxiliary invariants and lemmas. But this typically is much less effort-intensive than developing full correctness proofs using interactive theorem provers such as PVS.

The compression function implemented by the CMs has been modeled and analyzed using the calendar-automata framework [19]. The models includes a single CM, receiving frames from N SMs, out of which k may be faulty. Several properties of the compression functions have been verified on such models, but scalability issues limited the analysis to $N \leq 7$.

A more recent example of analysis focused on bounding the quality of the clock synchronization in TTEthernet for a network with two CMs and five SMs, and for different fault assumptions [20]. This analysis did not model the CMs clocks, but it focused on bounding the worst-case difference between the clocks of two SMs in the network, under different fault scenarios. As discussed previously, the verification relied on `sal-inf-bmc` and Yices, and required very little human guidance. A single invariant was added to improve scalability of the verification.

3.3 An Attempt at Combining Verification Results

Our goal when starting this work was to combine relevant verification results that were available, and build a coherent end-to-end proof of correctness of the TTEthernet clock-synchronization protocol. Our plan was to use SRI's ETB to support this combination. We intended to build upon a general model of fault-tolerant clock synchronization such as the PVS specification of the unified algorithm defined in [10], and cast the TTEthernet protocol as an instance of this generic scheme. We hoped to be able to discharge the proof obligations resulting from this instantiation using either PVS or SAL. These proof obligations amount to showing that the convergence functions implemented by TTEthernet satisfy assumptions made by the generic scheme.

However, we encountered several difficulties. First, the ETB is still being developed and is constantly being revised and improved. Previous prototype implementations of ETB do exist but they appear to be too limited for our purpose. The current status of the ETB development and an overview of its current architecture and design are presented the Appendix (A).

Another difficulty is that TTEthernet is not a simple protocol, and we did not manage to map it cleanly into an instance of the unified fault-tolerant algorithm [10]. One significant difficulty is that the TTEthernet clock synchronization is not divided into a succession of rounds, one of the key underlying assumption of the unified algorithm. Superficially, it may look like the TTEthernet clock synchronization is a two-round protocol: in round 1, SMs transmit PCFs to CMs; in round 2, CMs send compressed PCFs to all nodes. However, the compression function is actually more complicated than this would suggest. The compression

function is executed by a CM based on the PCFs it receives from SMs, but the function is independent of the CM's local clock. As a consequence, it may be possible in some scenarios for one CMs to execute multiple instances of the compression function in one integration cycle, and this may potentially result in several compression PCFs being sent by the same CM during one integration cycle.

Although this kind of behavior should not happen under normal circumstances (i.e., when the nodes are synchronized), it is dangerous to rule it out a priori. We should prove that multiple compressed PCFs per round can't occur if the network is synchronized, rather than building a model in which multiple compressed PCFs cannot occur and use this model to prove that the clock synchronization works.

A further issue is the mismatch between the mathematical modeling used in the existing proofs of clock-synchronization protocols developed in PVS (or other theorem provers), and the state-machine model required for analysis with SAL. There may be abstraction techniques or other approaches to bridging the gap between abstract, mathematical models of clock synchronization developed in PVS and the more concrete and operational, state-machine models that are specified in SAL. However, we have not found such a bridge yet.

3.4 Toward Unified State-Machine Models

As clearly illustrated by the existing SAL models of TTEthernet fragments, state-machine models are extremely useful. They can be quickly analyzed using model checkers or other methods such as simulation. They can be used to guide the design of complex protocols. A state-machine description is a good match for most distributed protocols, and state machines can be used for automated test generation.

However, state-machine verification using model checking has well known limitations. Typically, a state-machine model must be limited to finite (and usually small) protocol instances. We can't expect current model-checking techniques to deliver a full, general proof of the correctness of TTEthernet clock synchronization. In addition, even if we are interested in only a fixed set of instances, model checking still suffers from the state-explosion problem or other scalability issues. Ultimately, any full formal proof of correctness for protocols as complex as the TTEthernet clock synchronization requires interactive theorem proving.

Our plan is to develop state-machine models that can be conveniently analyzed with model checking tools such as SAL, possibly for only partial verification, but can also be easily translated to PVS specifications in a form that enables full-correctness. In future work, we hope to support a common state-machine notation supported by the ETB with appropriate conversions to enable analysis using a variety of verification tools.

As a first step toward this goal, we are currently exploring a new approach for specifying and verifying the TTEthernet clock synchronization protocol as a state machine in PVS. We outline the results we have obtained so far.

4 A New PVS Formalization of TTEthernet

As discussed previously, our goal is to develop a PVS model of the TTEthernet clock-synchronization protocol that is based on a state-machine paradigm. We want to model the CMs, SMs, and SCs as state transition systems, so that their specification stays close to what can be analyzed using SAL. Building and verifying state-machine models of distributed systems is very common in interactive theorem proving, but it is usually applied to discrete systems. Adding real time, transmission delays, node failure, and clock drifts significantly increases the model complexity.

Our main challenge is to obtain a model that remains amenable to PVS verification without excessive effort. The verification effort required should not be worse than what was required for the existing more axiomatic specifications that exits in PVS.

4.1 Modeling Approach

The modeling approach we are currently exploring builds on the timeout and calendar automata concepts that were originally intended for SAL, and have been successfully applied to various distributed timed systems, including TTEthernet [19, 20] and other timed-triggered protocols [5].

Both timeout and calendar automata are discrete state-transition systems, where some real-valued state variables are used to model time progress and specify timing constraints. In general, a state-transition system is a triple of the form $\langle S, I, \rightarrow \rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, and $\rightarrow$ is a binary relation on S called the *transition relation*. We assume that the state space S is built from a collection of state variables: each state σ of S is a mapping that assigns a value of an appropriate type to each of the system's state variables. For example, if x is a state variable and σ is a state of the system, then $\sigma(x)$ denotes the value of variable x in state σ.

A *timeout automaton* is a traditional state-transition system equipped with a finite set T of special-purpose variables that model timeouts. The system also includes a dedicated state variable t that stores global time. The initial states and transition relation must satisfy the following requirements:

- In any initial state σ, we have $\sigma(t) \leq \sigma(x)$ for all $x \in T$.
- If σ is a state such that $\sigma(t) < \sigma(x)$ for all $x \in T$ then the only transition enabled in σ is a *time progress transition*. It increases t to $\min(\sigma(T)) = \min\{\sigma(x) \mid x \in T\}$ and leaves all other state variables unchanged.
- Discrete transitions $\sigma \rightarrow \sigma'$ are enabled in states such that $\sigma(t) = \sigma(x)$ for some $x \in T$ and satisfy the following conditions
  - $\sigma'(t) = \sigma(t)$
  - for all $y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma'(t)$
  - there is $x \in T$ such that $\sigma(x) = \sigma(t)$ and $\sigma'(x) > \sigma'(t)$.

In all reachable states, a timeout x never stores a value in the past, that is, the inequality $\sigma(t) \leq \sigma(x)$ is an invariant of the system. A discrete transition can be taken whenever the time t reaches the value of one timeout x. Such a transition must increase at least one such x to a time in the future, and if it updates other timeouts than x their new value must also be in the future. Whenever the condition $\forall x \in T : \sigma(t) < \sigma(x)$ holds, no discrete transition is enabled and time advances to the value of the next timeout, that is, to $\min(\sigma(T))$. Conversely, time cannot progress as long as a discrete transition is enabled.

Discrete transitions are instantaneous since they leave t unchanged. Several discrete transitions may be enabled in the same state, in which case one is selected nondeterministically. Several discrete transitions may also need to be performed in sequence before t can advance, but the constraints on timeout updates prevent infinite zero-delay sequences of discrete transitions.

In typical applications, the timeouts control the execution of n real-time processes $p_1, \ldots, p_n$. A timeout x_i stores the time at which the next action from p_i must occur, and this action updates x_i to a new time, strictly larger than the current time t, where p_i will perform another transition. For example, we have used timeout-based modeling for specifying and verifying Fischer's mutual exclusion algorithm [6].

Calendar automata extend the previous *timeout automata* model with support for modeling communication between processes, including constraints on message latency and other delays. This is achieved by adding *event calendars* to the transition system.

A calendar is a finite set (or multiset) of the form $C = \{\langle e_1, t_1 \rangle, \ldots, \langle e_n, t_n \rangle\}$, where each e_i is an event and t_i is the time when event e_i is scheduled to occur. All t_is are real numbers. We denote by $\min(C)$ the

smallest number among $\{t_1, \ldots, t_n\}$ (with $\min(C) = +\infty$ if C is empty). Given a real u, we denote by $\mathrm{Ev}_u(C)$ the subset of C that contains all events scheduled at time u:

$$\mathrm{Ev}_u(C) \quad = \quad \{\langle e_i, t_i\rangle \mid t_i = u \ \wedge \ \langle e_i, t_i\rangle \in C\}$$

As before, the state variables of a calendar-based system $\mathcal{M}$ include a real-valued variable t that denotes the current time and a finite set T of timeouts. In addition, one state variable c stores a calendar. These variables control when discrete and time-progress transitions are enabled, according to the following rules:

- In all initial state σ, we have $\sigma(t) \leq \min(\sigma(T))$ and $\sigma(t) \leq \min(\sigma(c))$.
- In a state σ, time can advance if and only if $\sigma(t) < \min(\sigma(T))$ and $\sigma(t) < \min(\sigma(c))$. A time progress transition updates t to the smallest of $\min(\sigma(T))$ and $\min(\sigma(c))$, and leaves all other state variables unchanged.
- Discrete transitions can be enabled in a state σ provided $\sigma(t) = \min(\sigma(T))$ or $\sigma(t) = \min(\sigma(c))$, and they must satisfy the following requirements:
  - $\sigma(t) = \sigma'(t)$
  - for all $y \in T$ we have $\sigma'(y) = \sigma(y)$ or $\sigma'(y) > \sigma'(t)$
  - if $\sigma(t) = \min(\sigma(c))$ then $\mathrm{Ev}_{\sigma'(t)}(\sigma'(c)) \subseteq \mathrm{Ev}_{\sigma(t)}(\sigma(c))$
  - we have $\mathrm{Ev}_{\sigma'(t)}(\sigma'(c)) \subset \mathrm{Ev}_{\sigma(t)}(\sigma(c))$, or there is $x \in T$ such that $\sigma(x) = \sigma(t)$ and $\sigma'(x) > \sigma'(t)$.

These constraints ensure that $\sigma(t) \leq \min(\sigma(T))$ and $\sigma(t) \leq \min(\sigma(c))$ are invariants: timeout values and the occurrence time of any event in the calendar are never in the past. Discrete transitions are enabled when the current time reaches the value of a timeout or the occurrence time of a scheduled event. The constraints on timeout are the same as before. In addition, a discrete transition may add events to the calendar, provided these new events are all in the future. To prevent instantaneous loops, every discrete transition must either consume an event that occurs at the current time or update a timeout as discussed previously.

Calendars are useful for modeling communication channels that introduce transmission delays. An event in the calendar represents a message being transmitted and the occurrence time is the time when the message will be received. The action of sending a message m to a process p_i is modeled by adding the event "p_i receives m" to the calendar, which is scheduled to occur at some future time. Message reception is modeled by transitions enabled when such event occurs, and whose effects include removing the event from the calendar. From this point of view, a calendar can be seen as a set of messages that have been sent but have not been received yet, with each message labeled by its reception time.

The PVS model we are developing for TTEthernet is largely based on the calendar automata formalism. However, we are planning to extend the notion of calendar (i.e., a state variable that stores future events) into a more general model, where both past and future events are part of the system state. Thus, the system model includes a general event set that includes both all events that have occurred in the past (i.e., similar to a history or trace-based model) and events that are scheduled to occur in the future (i.e., pending events such as message reception). We believe keeping track of past events as part of the system state will facilitate formalization and verification of timing properties, as all potentially relevant events are part of the state and labeled with the time at which they occur.

4.2 Current Status

The whole PVS theories that we have developed for modeling TTEthernet are available at `http://www.csl.sri.com/vvfcs.html`. The developments so far have focused on defining and proving several properties of the TTE compression function. This required developing supporting theories, that define several variants of orderings and sorting functions, including sorting of finite sets and finite vectors.

The most notable result we have proved so far is the fact that the TTE compression function has the expected convergence property, except in a somewhat surprising special case. Relevant fragments of the PVS formalization are shown in Figure 3. The core of the compression algorithm is the function `compress` shown in the figure. This function takes a finite vector `v` of values as input, and computes an average of the vector components. The actual averaging operation applied depends on the size of the vector. For example, if `v` contains three, four, or five elements, `compress` returns the median of these elements.

A CM applies this `compress` function to a set of PCFs it receives from SMs. This requires first sorting these PCFs and computing clock differences. The details of the full procedure are not shown in the figure.[2] The operation is denoted by `comp_correction(B)` in Figure 3, where B is a finite set of PCF frames.

The convergence result is proposition `similar_synchronized_convergence` in Figure 3. Informally, this result states that the compression function makes clocks get closer together. If two compression masters apply the compression function to two distinct sets of PCFs `C1` and `C2` that come from approximately synchronized SMs, then the resulting compression values are close (no more than a bound equal to half the precision Δ plus a small error term). Thus the compression function essentially reduces the clock skew by half. This property holds under various constraints about the number of good values included in `C1` and `C2`, in relation to `K`, the number of faulty SMs to tolerate.

What is somewhat surprising is that convergence is not guaranteed if both `C1` and `C2` contain exactly five PCFs. This is a consequence of the definition of `compress`. If both `C1` and `C2` have five elements, the compression function will pick the median in both sets. If there is a Byzantine faulty SM, then `C1` and `C2` contain four good PCFs and one other input from the faulty SM. Depending on where the Byzantine values lie in `C1` and `C2`, the medians may be different and, in the worst case, the clock skew does not improve.

In the next section, we examine how this special case affects the quality of the clock synchronization between CMs. This analysis is based on a SAL model of the protocol. As can be anticipated, the fact that convergence does not always hold has an impact on the worst-case clock skew between CMs. The network remains globally synchronized in the sense that the worst-case clock difference between two network components is bounded, but the maximal clock skew is worst for the CMs than for the SMs.

We also note that a simple change to the compression function will get rid of this issue, namely, replacing the median by a fault-tolerant midpoint for `m=5`:

```
compress(m)(v: cvector(m)): clock_time =
     ...
     m = 5 -> avg(v(1), v(3)),
     ...
```

5 SAL Analysis of the Compression Function

The PVS verification helped us uncover a possible flaw in the compression function used by TTEthernet. We now investigate the impact of this flaw and the proposed fix on the quality of the clock synchronization. For this purpose, we build a (simplified) SAL model of the resynchronization protocol and we compute bounds on the worst-case clock drift between different network components. This SAL model generalizes a previous formalization presented in [20], which focused on bounding the clock drift between distinct SMs.

[2] See `http://www.csl.sri.com/users/bruno/vvfcs.html` for the full PVS specifications.

```
%
% Compression function for a vector v
% - v must have at least one element
%
m: VAR posnat

compress(m)(v: cvector(m)): clock_time =
  COND
    m = 1 -> v(0),
    m = 2 -> avg(v(0), v(1)),
    % for m = 3, 4, 5: result = median
    m = 3 -> v(1),
    m = 4 -> avg(v(1), v(2)),
    m = 5 -> v(2),
    % for m>5: result =
    % average of (k+1)-th smallest and (k+1)-th largest elements
    ELSE  -> avg(v(K), v(m-K-1))
  ENDCOND

...

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)

...

%
% Main result: if C1 and C2 contain readings from good SMs form I,
% and the cardinality constraints below are satisfied then
%  | comp_pit(C1) - comp_pit(C2) | <= precision/2 + 1 + eps
%
% Because of the non-uniform definition for m1 = 5 or m2 = 5,
% we don't get convergence if both are equal to 5.
%
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 3. Compression Function in TTEthernet

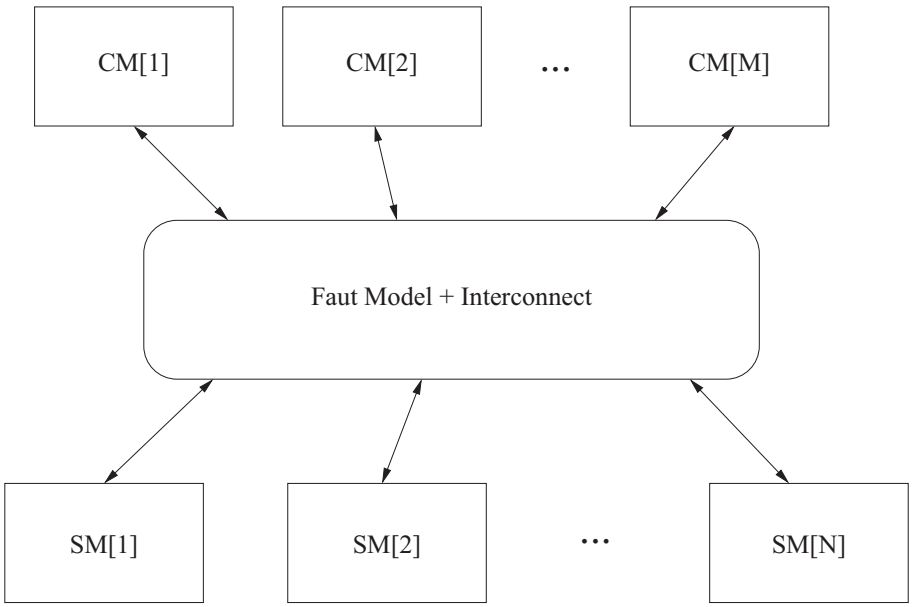

Figure 4. SAL Model Structure

The SAL model used then represented the CMs as stateless components and did not include CM clocks. The new formalization represents both SMs and CMs as state machines with real-valued clocks, which enables us to bound the clock difference between two CMs and between a CM and an SM. We have also revised significant parts of the SAL specifications to increase modularity, and we use a simpler formalization of the compression function.

5.1 SAL Model

In our model of the TTEthernet clock synchronization protocol, all the participants (i.e., CMs and SMs) are modeled as SAL state machines. Each state machine has a real-valued state variable that models the component's local clock. We decouple behavior and fault modeling by following the abstraction suggested by Pike et al. [12]:

- All components are modeled as if they were fault free.
- All faults are represented as communication faults.

Under this abstraction, all components behave correctly, but faults affect the ability of components to transmit data. This abstraction is general enough to capture typical fault assumptions, since a fault is typically characterized by its manifestation on a component's interface (cf. [12]).

More concretely, the SAL model is the composition 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 (Fig. 4). Faults are then modeled in the interconnect. If a source process is nonfaulty, 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.

```
POSREAL: TYPE = { x:REAL | x > 0 };
max_drift: POSREAL;

CLOCK: TYPE = REAL;

%
% N = number of Synchronization Masters (SMs)
% M = number of Compression Masters (CMs)
%
N: NATURAL = 5;
M: NATURAL = 2;

SM_ID: TYPE = [1 .. N];
CM_ID: TYPE = [1 .. M];
```

Figure 5. SAL Types and Parameters

5.1.1 Types and Parameters

Figure 5 shows several types and parameters used in our SAL formalization. The constant `max_drift` represents the maximal drift that a clock can experience in one integration cycle. We model clocks as real-valued variables (of type `CLOCK`). The model is a fixed TTEthernet instance that consists of `N` synchronization masters and `M` compression masters. The particular instance shown in Figure 5 has five SMs and two CMs. Each SM is identified by an index of type `SM_ID` (i.e., an integer between 1 and `N`). Similarly, each CM is identified by an index of type `CM_ID` between 1 and `M`.

5.1.2 Synchronization Master

The SAL model of a synchronization master is shown in Figure 6. Each SM is assumed to receive one compression message from each CM. This is modeled as an input variable `compression`, which is an array of M clock values. The main output is the SM's `clock` variable. An SM cycles through three successive states: `sm_send`, `sm_correct`, and `sm_drift`. The first two states correspond to the two-phase synchronization protocol of TTEthernet that is executed at the beginning of every integration cycle. In state `sm_send`, the SM transmits its clocks to the two CMs; in state `sm_correct`, the SM reads one compression value from each CM and corrects its clock by computing the average of the two values.

State `sm_drift` abstracts the rest of an integration cycle. In this state, the SM's clock drifts from real time by some amount bounded by the `max_drift` parameter. The variable `clock` is then the positive or negative drift of the SM's clock relative to real time. This encoding based on drift instead of absolute clock time is similar to the one used in [20].

5.1.3 Compression Master

The SAL model of compression masters is similar to the SM model. A CM cycles through three successive states representing three different phases of an integration cycle. In state `cm_receive`, a CM reads inputs from the SMs and outputs a `compression` value. In state `cm_correct`, the CM applies a clock correction: it resets its clock to the compression value computed in the previous state. Then in state `cm_drift`, the CM clock drifts from real time by some amount bounded by `max_drift`.

Since we are interested in scenarios involving faulty SMs, we must take into account the possibility that clock values received by a CM may be incorrect or missing. For this purpose, we model the input to a CM as two variables: `sm_reading` and `sm_valid`. Variable `sm_reading` is an array of N clock values

```
SM_STATE: TYPE = { sm_send, sm_correct, sm_drift };

SM: MODULE =
  BEGIN
    INPUT
      compression: ARRAY CM_ID OF CLOCK

    OUTPUT
      state: SM_STATE,
      clock: CLOCK

    INITIALIZATION
      state = sm_send;
      clock = 0;

    TRANSITION
      [ state = sm_send -->
          state' = sm_correct;

     [] state = sm_correct -->
          state' = sm_drift;
          clock' = (compression[1] + compression[2])/2; % correction

     [] state = sm_drift -->
          clock' IN { x : CLOCK | clock - max_drift <= x AND x <= clock + max_drift };
          state' = sm_send;
     ]
   END;
```

Figure 6. SAL Model of a Synchronization Master

and sm_valid is an array of N Boolean values. Given an SM index i, variable sm_valid[i] specifies whether or not a message was received from SM i, and, if sm_valid[i] is true, sm_reading[i] is the value in this message.

Defining the compression function in SAL is not as easy as in PVS, as the definition requires sorting the clock readings in increasing order. SAL supports recursive definition, so in principle it is possible to define a sort function in SAL. However, any recursive definitions of a sort function is very expensive for SAL to process. A better approach is to define a sort predicate as follows:

```
sort(c: ARRAY SM_ID OF CLOCK, p: ARRAY SM_ID OF SM_ID): BOOLEAN =
      (FORALL (i: SM_ID): i<N => c[p[i]] <= c[p[i+1]])
  AND (FORALL (i, j: SM_ID): p[i] = p[j] => i = j);
```

This definition specifies that sort(c, p) is true if the array p is a permutation of the SM indices in $\{1, \ldots, N\}$, and the sequence c[p[1]], ..., c[p[N]] is increasing. This specification trick was introduced in [19]. It works well as long as the number of clock readings to sort is equal to the parameter N. In our new SAL model, we must deal with a more complex situation. Some clock readings are marked as invalid (i.e., missing) and the compression function requires sorting the array of valid clock readings in increasing order, while ignoring the invalid elements. In the SAL model used in [20], we defined a separate predicate for partial sort to deal with the case where one input was missing. This resulted in a relatively complicated SAL specification involving two different sort predicates and extra logic and variables to select between both. This approach was also hard to extend to other scenarios such as two missing clock readings.

We are now using a more uniform and simpler specification, which relies on the following definition.

```
sort(c: ARRAY SM_ID OF CLOCK, v: ARRAY SM_ID OF BOOLEAN,
     n: [0 .. N], p: ARRAY SM_ID OF SM_ID): BOOLEAN =
        (FORALL (i: SM_ID): i<n => c[p[i]] <= c[p[i+1]])
   AND (FORALL (i: SM_ID): v[p[i]] <=> (i <= n))
   AND (FORALL (i, j: SM_ID): p[i] = p[j] => i = j);
```

In this definition, c is an array of N clock values, and v is a Boolean array that indicates which clock values are valid. The extra parameter n is intended to denote the number of valid elements in c (or, equivalently, the number of true elements in v). Then, the definition states that predicate sort(c, v, n, p) holds if the following conditions are satisfied:

- There are exactly n valid values in c
- p is a permutation of the SM indices in $\{1, \ldots, N\}$
- The sequence p[1], ..., p[n] lists the valid value indices (i.e., v[p[1]],...,v[p[n]] are all true and v[p[n+1]], ..., v[p[N]] are all false).
- The sequence of clock values c[p[1]],...,c[p[n]] is increasing.

In other words, the sequence p[1],...,p[n] enumerates the valid elements of c in increasing order.

Using this definition, we can define the compression function as shown in Figure 7. The CM module includes an auxiliary state variable perm (intended to store a permutation of the SM indices) and different guarded commands define the compression values based on the number of valid clock readings. For example, the case where five good values are received is specified as:

```
[] state = cm_receive AND sort(sm_reading, sm_valid, 5, perm') -->
      state' = cm_correct;
      compression' = sm_reading[perm'[3]];
```

This definition is possible because SAL allows us to refer to perm' (that is, the value of perm in the next state) in the guard. This definition is simpler, more general, and more concise than the one we used in [20]. It also improves SAL performance as it reduces the number of state variables and does not require as much case analysis.

```
CM_STATE: TYPE = { cm_receive, cm_correct, cm_drift };

CM: MODULE =
  BEGIN
    INPUT
      sm_reading: ARRAY SM_ID OF CLOCK,
      sm_valid: ARRAY SM_ID OF BOOLEAN

    LOCAL
      perm: ARRAY SM_ID OF SM_ID

    OUTPUT
      state: CM_STATE,
      clock: CLOCK,
      compression: CLOCK

    INITIALIZATION
      state = cm_receive;
      compression = 0;
      clock = 0;

    DEFINITION
      perm IN { p: ARRAY SM_ID OF SM_ID | TRUE };

    TRANSITION
      [ state = cm_receive AND sort(sm_reading, sm_valid, 3, perm') -->
          %% received 3 clock readings (i.e., two faulty SMs)
          state' = cm_correct;
          compression' = sm_reading[perm'[2]];

     [] state = cm_receive AND sort(sm_reading, sm_valid, 4, perm')  -->
          %% received 4 clock readings
          state' = cm_correct;
          compression' = (sm_reading[perm'[2]] + sm_reading[perm'[3]])/2;

     [] state = cm_receive AND sort(sm_reading, sm_valid, 5, perm') -->
          %% received 5 clock readings
          state' = cm_correct;
          compression' = sm_reading[perm'[3]];

     [] state = cm_correct -->
          clock' = compression;
          state' = cm_drift;

     [] state = cm_drift -->
          clock' IN { x : CLOCK | clock - max_drift <= x AND x <= clock + max_drift };
          state' = cm_receive;
      ]
  END;
```

Figure 7. SAL Model of a Compression Master

5.1.4 Interconnect and Fault Model

The rest of the SAL model defines assumptions about faulty components and communication. We first define a function `status` that indicates which components may be faulty and the type of faults they may suffer. For example, in a scenario with one Byzantine faulty SM out of five, we could write:

```
STATUS: TYPE = { good, omissive, byzantine };

sm_status(i: SM_ID): STATUS = IF i=3 THEN byzantine ELSE good ENDIF;
```

In this particular example, we have assumed that `SM[3]` is faulty. Any other choice of index for the faulty SM makes no difference to the analysis, and it's also possible to assert that one SM is faulty without specifying which one (but this makes the analysis slightly less efficient).

Based on the `sm_status` function defined above, we use an interconnect module that specifies how the output value from each SM is received by each CM. The interconnect module is shown in Figure 8. The input to this module is an array of N clock values, namely, the clock variables of each SM. The interconnect's output consists of two arrays `sm_reading` and `sm_value` of dimension M: `sm_reading[j]` and `sm_valid[j]` are the two input to module `CM[j]`. They are themselves arrays of N elements. In other words, the element `sm_reading[j][i]` is the clock value received by `CM[j]` from `SM[i]`, and `sm_valid[j][i]` indicates whether this value is valid or missing. The fault model is then expressed by the following rules:

- If `SM[i]` is nonfaulty then

  `sm_reading[j][i] = sm_clock[i]` and `sm_valid[j][i] = true`

- If `SM[i]` omissive[3] faulty then

  `sm_reading[j][i] = sm_clock[i]` and `sm_valid[j][i]` can be either true or false.

- If `SM[i]` is Byzantine then

  both `sm_reading[j][i]` and `sm_valid[j][i]` are arbitrary.

These rules are written in the SAL syntax in Figure 8.

5.2 Properties and Analysis

The complete SAL models we have used for analyzing the compression function are available at `http://www.csl.sri.com/users/bruno/vvfcs.html`. Three variant models are included:

- The baseline model uses the compression function as defined in the TTEthernet standard. It describes a configuration with five synchronization masters and two compression masters, with the assumption that one synchronization master is Byzantine faulty.

- A variant model also uses the standard's compression function but examines a different configuration and fault scenario. This model includes six synchronization masters and two compression masters, and assumes that two synchronization masters are omissive faulty.

- The last model uses the same configuration and fault assumption as the baseline—five SMs, two CMs, one Byzantine faulty SM— but the compression function uses the fix we propose in Section 4. When a CM receives five valid readings, it computes the compression value as the average of the second and fourth value instead of taking the median.

[3]Here omissive means "follows the inconsistent omission fault model".

```
Connection: MODULE =
  BEGIN
    INPUT
      sm_clock: ARRAY SM_ID OF CLOCK
    OUTPUT
      sm_reading: ARRAY CM_ID OF ARRAY SM_ID OF CLOCK,
      sm_valid: ARRAY CM_ID OF ARRAY SM_ID OF BOOLEAN

    DEFINITION
      %% sm_valid[j][i] is true if i is GOOD
      sm_valid IN { B: ARRAY CM_ID OF ARRAY SM_ID OF BOOLEAN |
                    FORALL (j: CM_ID, i: SM_ID):
                      sm_status(i) = good => B[j][i] };

      %% sm_reading[j][i] is equal to sm_clock[i] unless i is Byzantine
      sm_reading IN { A: ARRAY CM_ID OF ARRAY SM_ID OF CLOCK |
                      FORALL (j: CM_ID, i: SM_ID):
                         sm_status(i) /= byzantine => A[j][i] = sm_clock[i] };
  END;
```

Figure 8. Interconnect Module

In each model, we want to bound the maximal clock drift between network components. We illustrate how this can be done in the next section, where we establish the worst-case clock drift between two synchronization masters. We then summarize the bounds we have obtained in each model for the different types of components.

5.2.1 Analysis Method

Since the model assumes that the compression masters are not faulty (and makes other simplifying assumptions), it is easy to see that the SMs are closely synchronized with each other. The following invariant holds in all three SAL models:

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

Because individual clocks can drift from real time by $\pm$`max_drift` during each integration cycle, the bound `2 * max_drift` is the best that can be achieved in the models. We can easily check this in SAL by showing that the following stronger property does not hold:

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

A counterexample to the latter property can be found by `sal-inf-bmc`, using a command such as

```
sal-inf-bmc -v 4 tte_synchro sm_clock_distance_strict -it
```

This invokes `sal-inf-bmc` to search for counterexamples to the property using iterative deepening (indicated by option `-it`). The command-line argument `-v 4` sets the "verbosity level" and is optional. With these options, `sal-inf-bmc` finds a counterexample trace of length three.

Using a similar command, we can check that there are no counterexamples to `sm_clock_distance`. But we can also obtain a proof that `sm_clock_distance` is satisfied. The proof is by k-induction and relies on simple auxiliary invariants. First, we can prove the following three lemmas:

```
phase1: LEMMA TTE |-
   G(FORALL (i: SM_ID, j: CM_ID):
       sm_state[i] = sm_send <=> cm_state[j] = cm_receive);
```

```
phase2: LEMMA TTE |-
   G(FORALL (i: SM_ID, j: CM_ID):
       sm_state[i] = sm_correct <=> cm_state[j] = cm_correct);

phase3: LEMMA TTE |-
   G(FORALL (i: SM_ID, j: CM_ID):
       sm_state[i] = sm_drift <=> cm_state[j] = cm_drift);
```

These three invariants show that all modules are in lock step. Each of them can be proved as follows:

```
sal-inf-bmc -v 4 -i -d 2 tte_synchro phase1
```

This commands `sal-inf-bmc` to perform a proof by induction at depth $d = 2$, which succeeds. Then, using one of the three above lemmas, the bound on clock drift between SMs can be established by induction at depth $d = 2$:

```
sal-inf-bmc -v 4 -i -d 2 tte_synchro sm_clock_distance -l phase1
```

Thus, we have proved that the worst-case clock skew between two synchronization masters is twice the maximal drift parameter `max_drift`. The proof was given above for the baseline model (in file `tte_synchro.sal`) but it works for the other two SAL models.

5.2.2 Analysis Results

By using the general approach outlined previously, we have proved the properties listed in Figure 9 for the two SAL models that use the standard TTEthernet compression function. The figure shows six properties, organized in three groups of two lemmas. The first lemma in each pair is true and has been proved with `sal-inf-bmc`; it establishes an upper bound on the difference between the clocks of two components. The second lemma is false; counterexamples can be found using `sal-inf-bmc`, which shows that the upper bound given by the first lemma is precise.

In both models, the synchronization masters are synchronized within a bound equal to twice the maximal drift. The synchronization bound for the two compression masters is four times the maximal drift, and the difference between the clock of an SM and the clock of a CM is three times the maximal drift. It is interesting to note that the flaw in the compression function affects clock precision even in a scenario without Byzantine faults: the achievable precision in a scenario with six SMs, two of which are omissive faulty is the same as the precision with five SMs, one of which is Byzantine faulty.

The last SAL model relies on the revised compression function discussed previously, and the analysis confirms that this new compression function improves clock precision. The clocks of the two compression masters are guaranteed to be within `3 max_drift` of each other. and the worst case clock difference between an SM and a CM is now `2.5 max_drift`. The corresponding SAL properties are listed in Figure 10.

6 Conclusion

We have presented our work on formal analysis of the TTEthernet clock synchronization protocol using both PVS and SAL. Although this verification does not yet constitute a full end-to-end correctness proof, it enabled us to identify a suboptimal design choice in part of the current TTE standard, and to demonstrate how a simplex fix would improve clock precision.

In future work, we plan to develop a full PVS model that builds upon the calendar automata formalism. We hope this approach will allow us to develop of full correctness proof while relying on a modeling approach that is close to what can be verified using model checkers such as SAL.

```
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_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 4 * max_drift);

cm_clock_distance2_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 9. Clock Precision Achieved using TTE's Compression Function

```
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_distance2: LEMMA
   TTE |- G(FORALL (i, j: CM_ID): cm_clock[i] - cm_clock[j] <= 3 * max_drift);

cm_clock_distance2_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 10. Clock Precision Achieved with the Revised Compression Function

References

1. Astrit Ademaj and Hermann Kopetz. Timed-Triggered Ethernet and IEEE 1588 Clock Synchronization. In *IEEE Symposium on Precision Clock Synchronization (ISPCS 2007)*, pages 41–43. IEEE, 2007.

2. Damian Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verification of Clock Synchronization Algorithms: Experiment on a Combination of Deductive Tools. *Formal Aspects of Computing*, 19(3):321–341, August 2007.

3. Geoffrey M. Brown. Verification of a data synchronization circuit for all time. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 217–228, 2006.

4. Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

5. B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FORMATS/FTRTFT 2004)*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214. Springer-Verlag, September 2004.

6. Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International, July 2004. `http://www.csl.sri.com/user/bruno/publis/sri-sdl-04-03.pdf`.

7. Arvind Easwaran and Brendan Hall. Model-Driven Test Generation of Distributed System. Nasa contractor report, Honeywell, June 2011.

8. L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, January 1985.

9. J. Lundelius Welch and N. Lynch. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation*, 77:1–36, April 1988.

10. Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A Unified Fault-Tolerance Protocol. In *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 2004.

11. Paul S. Miner. Verification of fault-tolerant clock synchronization systems. Technical Paper 3349, NASA Langley Research Center, November 1993.

12. Lee Pike, Jeffrey Maddalon, Paul Miner, and Alfons Geser. Abstractions for Fault-Tolerant Distributed System Verification. In *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 257–270. Springer-Verlag, 2004.

13. John Rushby and Friedrich von Henke. Formal Verification of the Interactive Convergence Clock Synchronization Algorithm. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, February 1989. Revised August 1991.

14. Fred B. Schneider. Understanding protocols for byzantine clock synchronization. Technical Report 87-859, Cornell University, August 1987.

15. Natarajan Shankar. Mechanical verification of a schematic byzantine clock synchronization algorithm. Technical Report CR-4386, NASA, 1991.

16. Wilfried Steiner. An Evaluation of SMT-Based Schedule Synthesis for Timed-Triggered Multi-hop Nerworks. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS 2010)*, pages 375–384. IEEE Computer Society, 2010.

17. Wilfried Steiner. TTEthernet Executable Formal Specification. Deliverable for the CoMMiCS project, 2010.

18. Wilfried Steiner, Günter Bauer, Brendan Hall, Michael Paulitsch, and Srivastan Varadarajan. TTEthernet Dataflow Concept. In *Proceedings of the Eighth IEEE International Symposium on Network Computing and Applications (NCA 2009)*, pages 319–322. IEEE Computer Society, 2009.

19. Wilfried Steiner and Bruno Dutertre. SMT-Based Formal Verification of a TTEthernet Synchronization Function. In *Formal Methods for Industrial Critical Systems (FMICS 2010)*, volume 6371 of *Lecture Notes in Computer Science*, pages 148–164. Springer-Verlag, 2010.

20. Wilfried Steiner and Bruno Dutertre. Automated Formal Verification of the TTEthernet Synchronization Quality. In *NASA Formal Method Conference (NFM 2011)*, volume 6617 of *Lecture Notes in Computer Science*, pages 375–390. Springer-Verlag, 2011.

21. Wilfried Steiner, Michael Paulitsch, Brendan Hall, and Günter Bauer. The TTEthernet Synchronization Strategy: A Model-Driven Design Using SAL. Submitted for publication, 2009.

22. Wilfredo Torres-Pomales, Mahyar R. Malekpour, and Paul S. Miner. ROBUS–2: A Fault-Tolerant Broadcast Communication System. Technical Memorandum NASA/TM-2005-213540, NASA Langley Research Center, March 2005.

23. Timed-Triggered Ethernet. SAE Aerospace Standard, AS 6802, v1.1.2, 2011. draft.

24. William D. Young. Verifying the Interactive Convergent Clock-Synchronization Algorithm using the Boyer-Moore Prover. NASA Contractor Report 189649, NASA Langley Research Center, 1992.

Appendix A

The Evidential Tool Bus

The Evidential Tool Bus (ETB) is a framework for composing diverse inference tools in order to produce reproducible evidence supporting formal claims. The inference tools include theorem provers, SAT/SMT solvers, computer algebra systems, model checkers, code generators, and static and dynamic analyzers. Many formal demonstrations exploit a combinations of these tools. For example,

1. Counterexample-guided abstraction refinement (CEGAR) where a finite-state approximation (abstraction) of a system is analyzed by means of a model checker and counterexamples, if any, are used to refine the abstraction.

2. Concolic execution uses symbolic evaluation with a SAT or SMT solver to generate test inputs that drive a program along previously unexplored paths.

3. Bounded model checking employs a SAT or SMT solver to explore bounded length executions of a transition system.

4. Simplification using a computer algebra system such as REDUCE or QEPCAD to solve equations and perform quantifier elimination.

5. Proof obligation generation for pre/post-condition specifications and refinement steps using the PVS type checker.

6. Invariant generation using a range of techniques such as static analysis, templates, dynamic analysis, k-induction.

7. Combining a verification condition generator with a range of deductive techniques for discharging proof obligations.

8. Using a high-performance automated theorem prover to find an unsatisfiable core of input formulas that can then be checked with a proof-generating theorem prover.

The key benefit of ETB is to make it possible to rapidly prototype workflows that combine different analysis tools in order to generate claims that are accompanied by replayable evidence for an assurance case. The ETB merely serves as a bus for managing the flow of data and claims between different tools. It does not interpret any of the data or the claims. The ETB offers an interface for adding

1. New forms of data such as formulas, graphs, contexts, files, and test cases.

2. New judgement forms for making claims about the data.

3. Rules of inference for deriving claims from other claims.

4. New tools and tool interfaces.

The ETB can be used by itself to build and save proofs of claims built using the inference rules and tools mentioned above. It can also be employed as a back-end to other tools. In what follows, we examine the architecture of the ETB for managing data, claims, and rules, and for integrating analysis tools. While the ETB is focused on formal claims, the architecture can be adapted to other scenarios for tool integration.

Our first ETB prototype was built using XSB Prolog. The lessons learned from this exercise have been used in building a prototype using the Python scripting language. We outline a design for the architecture, API, and applications of ETB that is based on these experimental prototypes. In this design, scripts can be arbitrary code integrating multiple tools that deliver replayable proof scripts employing the available rules of inference and tool interfaces. Scripts can also employ backward and forward chaining through the rules of inference. For example, a procedure for generating an inductive invariant for a transition system produces an assertion that holds in the initial state and is preserved by each transition. Such a procedure might tie together multiple static and dynamic analysis tools, but once an invariant has been found, its inductivity can be checked by means of a simple proof script.

A.1 Data

Data in the ETB can be in the form of programs, transition systems, formulas, files, contexts, models, test cases, analysis results, The ETB processes data using external tools but has no built-in interpretation of the data. Many of the tools in the ETB are translators from one data representation to another. For example, an ETB tool can translate PVS or SAL formulas to the representation used by the Yices SMT solver. The data can either be represented as abstract syntax in XML or as part of a file. Some of the data might be handles that are understood by the component programs.

ETB data is either simple, e.g., integers or Booleans, or represented as a *blob* which is just a SHA-1 hash of the data. The data might have associated information that is kept in a hash table. This information indicates the *kind* of data, e.g., PVS formula, Yices context, or SAL module, as well as the file handle, or the abstract syntax representation. The ETB can be extended with new kinds of data.

A.2 Claims

The notion of a claim is central to ETB. The whole point of ETB is to produce a replayable proof script for a claim. This proof script incorporates calls to external tools. Claims are given in the form of judgements that are applied to data. Examples of judgements used to make claims include:

1. P is a PVS formula
2. Y is a Yices formula
3. Y is the Yices translation of PVS formula P
4. C is a counterexample for Y
5. I is a k-inductive property of machine M
6. Q is a Yices context representing the declarations D.
7. B is a CUDD BDD representation of Yices formula Y
8. Q is an unsatisfiable Yices context
9. A is the predicate abstraction of P with respect to the predicates Π

The labels B, I, P, Y, Q, and C range over data blobs. files or handles. Data that is handled within the toolbus itself will be in JSON or XML form. The tool bus has no built-in semantic interpretation of the claims. The ETB has an API for adding new claim forms. Each tool API is itself constitutes a claim. For example, checking the claim that P has type T in PVS context Q is done by invoking the PVS type checker.

A.3 Rules

The ETB rules of inference are based on Datalog, which is a simple form of logic programming developed for expressing database rules and queries. Datalog programs are logic programs where the terms are either variables or constants. In our case, the constants represent blobs and simple values such as Booleans, scalars, and integers, as well as, JSON arrays and objects. Some Datalog predicates correspond to specific tool invocations. We also associate names with individual rules in order to represent proofs.

For example, the claim of the invariance of an assertion P for a SAL module M using Yices for the inductivity check, is represented by the rules

1. *SalInvariantWithYices*:

$$\begin{aligned} &salInvariant(M, P) \text{ if} \\ &\quad salModule(M), \\ &\quad sal2Yices(M, M'), \\ &\quad sal2Yices(P, P') \\ &\quad yicesInductive(M', P') \end{aligned}$$

   An invariant property for a SAL transition system module can be checked by converting both the model and the property to a Yices state machine definition which is then verified using a rule shown below for checking state machine invariants.

2. *YicesInductivity*:

$$\begin{aligned} &yicesInductive(M, P), \text{ if} \\ &\quad yicesInitially(M, P), \\ &\quad yicesConsecution(M, P) \end{aligned}$$

   The rule for using Yices to check an invariant P for a state machine M reduces to checking that P holds initially and is preserved by each transition of M.

3. *YicesPredicateAbstract*:

$$\begin{aligned} &yicesPredicateAbstraction(V, S, P, A) \text{ if} \\ &\quad yicesNegation(P, Q), \\ &\quad yicesConjunction(S, Q, SQ), \\ &\quad yicesAllSMT(V, SQ, B), \\ &\quad yicesNegation(B, A) \end{aligned}$$

   The predicate abstraction of P with respect to a set of Boolean variables V that are mapped to predicates in S is constructed as A, by invoking an AllSMT interface for Yices on the negation of P and then negating the resulting Boolean formula.

The above rules contain calls to external tools for translating SAL transition systems to Yices state machines, and for invoking various Yices functions that are defined as part of the integration of Yices into the tool bus. Some of the ETB claims, such as those for checking the initiality and consecution of an invariant are themselves defined by further Datalog rules.

A.4 ETB Architecture

The ETB architecture consists of a claim table, a rule table, and Python scripts. Each row in the claim table consists of a partially instantiated claim with entries for

1. The status of the claim, i.e., if it is open, pending with subgoal claims, or closed (i.e., all subgoal claims are also closed).

2. The rule used to derive the claim from its subclaims.

3. The list of subgoals.

The claim table must satisfy the invariant that the claim must unify with the consequent of the rule, and the subgoal claims must be the corresponding instances of the antecedents of the rule. If the claim is closed, then the subgoal claims must also be closed. A closed claim must be fully instantiated, i.e., grounded.

The ETB API allows a range of operation on the claim table that preserve the invariant.

1. Open goal claims can be added. Claims can also be removed as long as there are no claims that depend on them.

2. Subgoal claims can be generated and added from goal claim by invoking a rule.

3. Goal claims corresponding to external calls can be executed. These external calls can also generate subclaim goals.

4. Proofs, both partial and completed, can be saved and reloaded.

A.5 Scripts in the ETB

By restricting the ETB core to the management of claims and (possibly partial) proofs, we can be flexible about what constitutes a script. Essentially, scripts manipulate the claims table while preserving the invariant. In particular, any program that produces a replayable proof in the claim table can be used as a script. Several scripts can be written by forward and backward chaining on the rules. For example, the rule *YicesPredicateAbstract* for predicate abstraction above can be directly used with backward chaining. On the other hand, an independent invariant generator can be used to produce an inductive invariant that is checked with the *YicesInductivity* rule.

In the earlier prototype, XSB Prolog was used as the scripting language, but this turned out to be quite awkward for writing scripts that involved simple iteration. Though XSB Prolog offers an excellent environment for prototyping, the kind of scripts we have in mind are more easily expressed in a dedicated scripting language like Python. For example, one such script for iterative bounded model checking searches for the smallest k such that a given assertion is k-inductive. The initiality check is a bounded model checking query to see if the invariant is violated in the initial k steps of the execution of a transition system. The consecution condition checks that if the assertion holds in a k-step execution sequence, than it is not violated in the $(k+1)$th step. Inductivity, as captured by the rule *YicesInductivity* is 0-inductivity. The script iterates from $k = 0$ upwards, which is quite easily expressed in Python but somewhat cumbersome to describe in Prolog.

A.6 Adding New Tools to the ETB

The basic way of invoking a tool or a tool API from ETB is by means of a goal claim. For example, `yicesUnsat`(P), can be associated with the invocation of the Yices SMT solver on the Yices formula P. Goal claims can contain variables. For example, the goal claim $yicesNegation(P, Q)$ when invoked on a

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 03 - 2012	Contractor Report	12/2010-07/2011

4. TITLE AND SUBTITLE

Integrated Formal Analysis of Timed-Triggered Ethernet

5a. CONTRACT NUMBER

NNL10AB32T

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Dutertre, Bruno; Shankar, Natarajan; Owre, Sam

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-2012-217554

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

This report was prepared by SRI International under NASA contract NNL10AB32T with Honeywell International, Inc., Golden Valley, MN.

Langley Technical Monitor: Paul S. Miner

14. ABSTRACT

We present new results related to the verification of the Timed-Triggered Ethernet (TTE) clock synchronization protocol. This work extends previous verification of TTE based on model checking. We identify a suboptimal design choice in a compression function used in clock synchronization, and propose an improvement. We compare the original design and the improved definition using the SAL model checker.

15. SUBJECT TERMS

Clock Synchronization; Fault Tolerance; Formal Methods; Time-Triggered Ethernet; Verification

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	33	(443) 757-5802