2011 17th IEEE Pacific Rim International Symposium on Dependable Computing

Layered Diagnosis and Clock-Rate Correction for the TTEthernet Clock Synchronization Protocol

Wilfried Steiner
Chip IP Design
TTTech Computertechnik AG
wilfried.steiner@tttech.com

Bruno Dutertre
Computer Science Laboratory
SRI International
bruno@csl.sri.com

***Abstract*—Fault-tolerant clock synchronization is the foundation of synchronous architectures such as the Time-Triggered Architecture (TTA) for dependable cyber-physical systems. Clocks are typically local counters that are increased with a given rate according to real time, and clock synchronization algorithms ensure that any two clocks in the system read about the same value at about the same point in real time. This is achieved by a clock synchronization algorithm that changes the current values of the clocks, the clocks' rate, or both.**

This paper presents a diagnosis algorithm and a clock-rate correction algorithm as layered services on top of the TTEthernet clock synchronization algorithm, which itself is a clock-state correction algorithm. We analyze the algorithms' properties and explore and understand their behavior using a bounded model checker for infinite data types.

We use our formal framework for both simulation and formal proof. To the best knowledge of the authors this has been the first time that formal methods, should they be theorem provers or model checkers, have been applied to the problem of rate-correction for fault-tolerant clock synchronization. Furthermore, the formal development process itself demonstrates how easily existing models can be utilized in the development of new algorithms and their formal verification.

I. Introduction

Dependable systems are omnipresent in our daily lives, and are becoming increasingly large and complex. As a consequence of this trend it is apparent that the correct development of such complex systems requires a sound architectural basis. In the absence of architectures we will either build systems of insufficient quality or will simply not be able to build systems beyond a certain level of complexity at all. The time-triggered architecture (TTA) [1] as developed at the Institut für Technische Informatik at the Vienna University of Technology is an extraordinary example of an architecture for dependable embedded systems. The TTA tremendously simplifies the development of dependable cyber-physical systems. It has been successfully applied in industries that demand a high level of determinism such as the avionics industry, in which predictability of system operation is a key property. TTP [2] and TTEthernet [3] are implementations of the TTA. TTP is applied, for example, in the new Boeing 787 Dreamliner, whereas TTEthernet has been selected for the NASA Orion Space Program. While the aerospace and space industries (as well as automotive and similar industries) are traditional areas for dependable systems, we also observe emerging areas with increasing dependability requirements. Examples include surgery robots in the medical area, datacenters in critical industries, as well as the smart grid that aims at decentralized energy production and efficient energy use. TTEthernet is currently being evaluated for several of these areas.

TTEthernet integrates synchronized and unsynchronized communication on the same physical network, i.e., time-triggered frames and event-triggered frames can coexist. The synchronized, time-triggered traffic relies on synchronized local clocks in the system and, therefore, TTEthernet specifies a fault-tolerant clock synchronization algorithm. This algorithm is a clock state-correction algorithm: the TTEthernet devices periodically exchange the current values of their local clocks and correct their clock values appropriately.

We have formally verified the TTEthernet clock synchronization algorithm and reported the verification procedure and results in [4]. This has been the first time that a fault-tolerant clock synchronization algorithm has been formally verified by model checking under the assumptions of clock drift and clock failures. The presented verification method is highly automated, which not only minimizes the probability of human error in the deduction of the correctness of the algorithm, but also works as formal framework that enables the holistic design of new algorithms running on top of TTEthernet. In this paper we present two such algorithms, a diagnosis algorithm and a clock rate-correction algorithm, as well as their design process and their formal verification. The main contribution of this paper is, thus, twofold: we introduce a new method for algorithm development and, by demonstrating the method, we present two new algorithms.

The diagnosis algorithm implements a simple version of an accusation protocol as used in NASA's SPIDER protocol [5]. TTEthernet devices that detect inconsistencies in the TTEthernet clock synchronization protocol report these inconsistencies to all devices in the system. Once a sufficiently high number of devices has accused a particular device of being faulty, this device is excluded from the clock synchronization protocol. Using our formal framework, we

978-0-7695-4590-5/11 $26.00 © 2011 IEEE
DOI 10.1109/PRDC.2011.36

can automatically prove the resulting quality improvement of the precision in the system. Serafini et al. [6] introduce application-level diagnosis algorithms, for which they discuss an implementation and provide formal correctness proofs. Besides some algorithmic difference, from the point of view of formal verification Serafini et al. prove their algorithms in a discrete time model and abstract from the underlying synchronization protocol, while our framework allows the integrated proof of the clock synchronization protocol together with diagnosis in a continuous time model.

Clock rate-correction algorithms, or rate-correction algorithms for short, not only periodically realign the values of the clock counters, but also change the rate of their increment. E.g., a node may diagnose that it always has to correct its local clock for about $+5\mu s$. This means that the update rate of this clock's counter is too low, or in other words, the clock ticks too slowly. In this case, a rate-correction algorithm would speed-up the clock with the aim that the next correction should be less than $5\mu s$. Of course, this assumes some stability of clock drift, which we discuss later in this paper.

Probably most prominently, the FlexRay communication protocol [7] specifies a rate-correction algorithm. Although our algorithm is similar to the FlexRay rate-correction approach, there are differences with respect to the underlying assumptions on topology and the clock-state correction algorithm. A combination of clock-state correction and clock-rate correction has been introduced by Kopetz et al. in [8] and analyzed by simulation and measurement. This approach elects a particular rate master, which is then used by the other nodes to align their rate to. The drawback of such an approach is, that in case of the failure of the rate master, a re-election is necessary. Our approach does not rely on a rate master.

We continue in Section II with a review of the TTEthernet clock synchronization algorithm. In Section III we recapture the formal proof of this algorithm and present how its formal model is re-used in our framework for simulation and formal verification of layered algorithms. Based on this framework we have developed several new algorithms. We introduce a diagnosis algorithm in Section IV and a clock rate-correction algorithm in Section V. We discuss these algorithms formally and give simulation results and formal proofs using our framework. Finally, we conclude in Section VI.

Due to space limitations we do not discuss the SAL models in detail. The models can be found online[1] and will be referenced on the SAL wiki.

II. TTEthernet Clock Synchronization

TTEthernet is an extension of the traditional Ethernet standard with services that guarantee deterministic delivery of time-critical messages. An example network with two redundant channels is depicted in Figure 1. As depicted, a TTEthernet network consists of end systems and switches, where end systems are connected to switches with bi-directional communication links. Switches may connect to each other thereby forming multi-hop connections between end systems. Each switch belongs to one and only one channel and in its simplest form a channel is formed by a single switch and the communication links to the end systems. For fault-tolerance reasons, a TTEthernet network implements redundant channels, e.g., two redundant channels as in Figure 1.

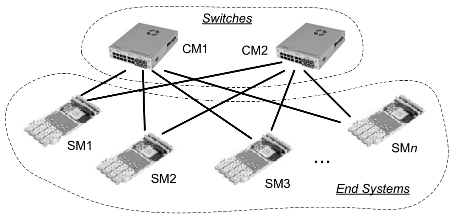

Figure 1: Example TTEthernet network with n end systems and two redundant channels (each formed by a single switch)

A. Clock Synchronization Overview

End systems and switches are physical components to which the TTEthernet clock synchronization algorithm assigns one of three "roles", synchronization master (SM), compression master (CM), or synchronization client (SC). In this paper we assume for simplicity that end systems are configured as SMs and switches as CMs. We also generally consider a network as the one depicted in Figure 1. SMs and CMs inform each other about the current state of their local clocks by exchanging protocol control frames (PCF). We have discussed the process of how a component concludes on the current local time of a remote component via the reception of PCFs in [9] and assume in the rest of the paper that the exchange of PCFs is equivalent to exchanging the current values of the local clocks of the components.

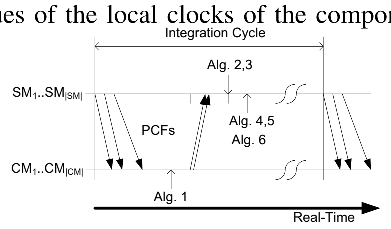

Figure 2: Overview of the TTEthernet clock synchronization algorithm

Figure 2 depicts the two steps in the TTEthernet clock synchronization algorithm. In the first step, the SMs send PCFs to the CMs. The CMs extract from the arrival points in time of the PCFs the current state of their local clocks and execute a first convergence function, the so-called compression function (Alg. 1). The result of the convergence function is then delivered to the SMs in the form of new

[1]`http://www.csl.sri.com/users/bruno/sal/layered-algorithms.tar.gz`

PCFs (the "compressed" PCFs). In the second step the SMs collect the compressed PCFs from the CMs and execute a second convergence function (Alg. 2,3). The diagnosis algorithm (Alg. 4,5) and the rate-correction algorithm (Alg. 6) analyzed in this paper are then executed only after the clocks are corrected by the TTEthernet clock synchronization algorithm.

B. *Failure Hypothesis*

TTEthernet assumes an inconsistent-omission failure model for the CMs. This means that a faulty CM is able to arbitrarily accept and reject PCFs from the SMs and can also decide to which SMs it sends the compressed PCF and to which not. Babbling idiot failures of the CM are excluded by the design of the CM as self-checking pair. The SMs, on the other hand, may fail arbitrarily, and in particular, they may start to babble PCFs. The design of the CMs ensures that only one PCF per SM is used per re-synchronization cycle. However, we assume that the clock values provided by a faulty SM can be arbitrary and the faulty SM may send different clock values to the different CMs. Although TTEthernet is configurable to tolerate multiple failures, we analyze and verify the new algorithms under a single failure fault-hypothesis. Hence, we assume either a faulty SM or a faulty CM, but not both at the same point in time.

C. *First Step Convergence: Compression Master (CM)*

The CMs collect the current states of the local clocks of the SMs. We denote these values by SM_clock_i, where $1 \leq i \leq |SM|$ and assume that the SM_clock_i values are sorted in increasing order. From the received SM_clock_i, a CM j uses a variant of the fault-tolerant median to calculate the new "compressed" clock CM_clock_j. Algorithm 1 defines this calculation as a function of the number of SM_clock_i values (denoted by the cardinality $|SM\_clock|$) received.

Algorithm 1 Convergence Algorithm executed by CM j

```
1: if $|SM\_clock| = 1$ then
2:     $CM\_clock_j \leftarrow SM\_clock_1$
3: else if $|SM\_clock| = 2$ then
4:     $CM\_clock_j \leftarrow \frac{SM\_clock_1 + SM\_clock_2}{2}$
5: else if $|SM\_clock| = 3$ then
6:     $CM\_clock_j \leftarrow SM\_clock_2$
7: else if $|SM\_clock| = 4$ then
8:     $CM\_clock_j \leftarrow \frac{SM\_clock_2 + SM\_clock_3}{2}$
9: else if $|SM\_clock| = 5$ then
10:    $CM\_clock_j \leftarrow SM\_clock_3$
11: else
12:    average of the $(k+1)^{th}$ largest and $(k+1)^{th}$ smallest
       clocks, where $k$ is the number of faulty SMs to be
       tolerated.
13: end if
```

The compressed clock is delivered back to the SMs in a new "compressed" PCF and the SMs are able to read the compressed clock value from the arrival point in time of the compressed PCF. This compressed PCF also contains the *pcf_membership_new* field in its payload. *pcf_membership_new* is a bitvector in which each bit is assigned to a unique SM. The CMs will set the bit of a SM, if the respective SM i has provided a local clock value SM_clock_i in the calculation of the most recent CM_clock_j and will clear the bit otherwise. The self-checking pair design of the CM guarantees that the compressed clock CM_clock_j and the *pcf_membership_new* vector are consistent. Hence, the design prevents a faulty CM from setting an arbitrary number of bits in *pcf_membership_new*.

Alternatively, for the case of five SM clock values, the CM may calculate the arithmetic mean of the second and fourth SM clock value for the compressed clock value.

D. *Second Step Convergence: Synchronization Master (SM)*

In the second step of the clock synchronization algorithm, the SMs receive the compressed PCFs, extract the compressed clock values from them, and correct their local clocks. In the fault-free case each SM receives *exactly* one compressed PCF per CM from which it extracts the compressed clock values CM_clock_j, where $1 \leq j \leq |CM|$ and we assume the CM_clock_j values are sorted in ascending order.

In the case of a faulty CM, an SM may receive *at most* one compressed PCF per CM (as the faulty CM may decide not to send its compressed PCF to some SMs). Furthermore, an SM will only use a compressed PCF in its convergence function if the *pcf_membership_new* field has at least *accept_threshold* of bits set. The value of *accept_threshold* is calculated using Algorithm 2: the SM searches for the maximum number of bits set ($bits()$) in any of the PCFs received from the CMs. The value of *accept_threshold* is then given by this maximum minus the configured number of tolerable faulty SMs.

Algorithm 2 select(CM_clock)

```
1: for $j = 1 \rightarrow |CM|$ do
2:     if $current\_max < bits(pcf\_membership\_new_j)$
       then
3:         $current\_max \leftarrow bits(pcf\_membership\_new_j)$
4:     end if
5: end for
6: $accept\_threshold \leftarrow current\_max - conf\_faulty\_SM$
7: return $\{CM\_clock| \quad pcf\_memberhip\_new$
                $\geq accept\_threshold\}$
```

The SM will discard a compressed PCF that has fewer than *accept_threshold* bits set in the *pcf_membership_new* field. This mechanism ensures that an SM excludes compressed PCFs that represent relatively low numbers of SM clocks. The *pcf_membership_new* vector is also used in other TTEthernet algorithms such as clique detection or startup as well as in network configurations that use more than one CM per channel. We do not discuss this functionality and configurations in this paper. For the analysis of the clock synchronization algorithm the description above is sufficient.

Under the assumption of one CM per channel and no more than three channels, the convergence function is described in Algorithm 3.

Algorithm 3 Convergence Algorithm executed by SM i

```
1: if $|select(CM\_clock)| = 1$ then
2:     $act\_corr \leftarrow SM\_clock_i - CM\_clock_1$
3:     $SM\_clock_i \leftarrow CM\_clock_1$
4: else if $|select(CM\_clock)| = 2$ then
5:     $act\_corr \leftarrow SM\_clock_i - \frac{CM\_clock_1 + CM\_clock_2}{2}$
6:     $SM\_clock_i \leftarrow \frac{CM\_clock_1 + CM\_clock_2}{2}$
7: else
8:     $\{|select(CM\_clock)| = 3\}$
9:     $act\_corr \leftarrow SM\_clock_i - CM\_clock_2$
10:    $SM\_clock_i \leftarrow CM\_clock_2$
11: end if
```

The *act_corr* value is an extension for the rate-correction algorithm discussed in Section V. It stores the current correction value and is updated in each integration cycle.

E. Synchronization Theorem

The maximum difference between any two local clocks SM_clock_i and SM_clock_j of non-faulty SMs, SM_i and SM_j, is called the $precision$, which is bounded and known.

Theorem 1.

$$\begin{aligned}\forall i, j \quad : \quad & SM\_clock_i > SM\_clock_j \Rightarrow \\ & SM\_clock_i - SM\_clock_j \leq precision\end{aligned}$$

Proof: Theorem 1 has been proven by model checking in [4] considering a faulty SM, a faulty CM, or both a faulty SM and a faulty CM. ■

III. Proof and Simulation Framework

Our proof and simulation framework is based on the bounded infinite-state model checker SAL (`sal-inf-bmc`). We build on the representation of the TTEthernet clock synchronization algorithm presented in [4]. We treat this formal model as "holistic view" in a sense that our framework does not rely on the output of these previous studies of the TTEthernet clock synchronization algorithm, but incorporates the previous models with the new algorithms.

The algorithms presented in this paper have been formalized in SAL [10] as state-transition system of the form $\langle S, I, \rightarrow \rangle$. Here, S defines the set of system states σ_i, I the set of initial system states with $I \subseteq S$ and $\rightarrow$ the set of transitions between system states. Each system state σ maps the variables to particular values according to their defined variable type. Furthermore, SAL supports structured modeling such that we can define the SM and CM functionality in encapsulated modules.

SAL provides several tools (symbolic, bounded, and bounded infinite-state model checking). While we experimented with all of them, we finally use the bounded infinite-state model checker `sal-inf-bmc` to prove the TTEthernet synchronization quality as well as to generate testcases. With `sal-inf-bmc` we can treat time as a continuous entity and can use k-induction [11] as proof method.

A. Representation of Time

In TTEthernet, the clock synchronization algorithm is executed in rounds, called integration cycles. Figure 3 shows an example scenario with a fast and a slow clock.

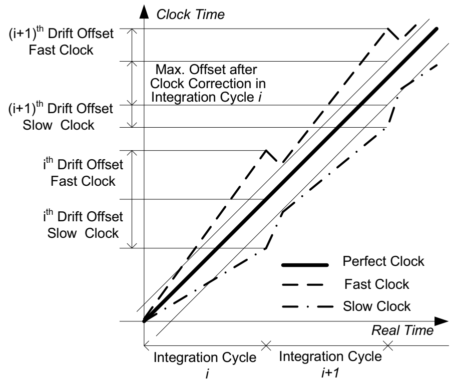
Clock Time
(i+1)th Drift Offset Fast Clock
Max. Offset after Clock Correction in Integration Cycle i
(i+1)th Drift Offset Slow Clock
ith Drift Offset Fast Clock
ith Drift Offset Slow Clock
Perfect Clock
Fast Clock
Slow Clock
Real Time
Integration Cycle i
Integration Cycle i+1

Figure 3: Progress in Real Time plotted against Clock Time

The x-axis depicts real time and the y-axis the internal clock time of a respective TTEthernet device. The perfect clock is plotted as a forty-five degree solid line while the fast clock is depicted as a dashed line slightly above the perfect clock and the slow clock is depicted as a dotted-dashed line slightly below the perfect clock. The figure shows for each integration cycle the divergence of the fast and slow clocks from the perfect clock and their synchronization at the beginning of each integration cycle. The drift from the perfect clock is a function of the length of the integration cycle and the drift rate of the clocks. Following literature we use R_{sync} for the integration cycle and ρ for the drift rate. In addition, we use a value Δ_{error} to summarize other factors in the clock synchronization process (e.g., network jitter, inaccuracies from the clocks not perfectly executing the integration cycles at the same time), but in general we assume that Δ_{error} will be a rather small factor compared to the real clock drift. Hence, we use the term drift offset, or *drift* for short, for the sum of deviations of a clock from the perfect clock within one integration cycle.

$$drift = R_{sync} \times \rho + \Delta_{error} \tag{1}$$

In TTEthernet we are interested in the precision of the non-faulty clocks, where the precision is defined as the maximum difference between any two non-faulty clocks in the system. Now, in order to determine the precision we do not need the actual clock readings, but only the sequences of their differences to the perfect clock. Figure 4 shows this approach of modeling time to verify the precision. The *x*-axis represents real time, the *y*-axis represents the clock time deviations from the perfect clock. In each step from even to odd *x* values the drift offset for an integration cycle i is added. At each even *x* value we see the maximum offset after the execution of the clock synchronization algorithm. Note that the *x*-axis is therefore not equally spaced with respect to real time, the drift offset step simulates the drift over the integration cycle, while the execution time of the clock synchronization algorithm is only a fraction of the integration cycle length.

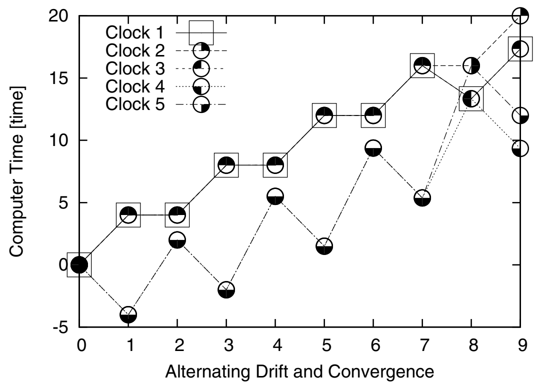

Figure 4: Example execution of the TTEthernet clock synchronization algorithm in presence of a faulty CM.

B. Simulation with SAL

The design of distributed algorithms is notoriously difficult, in particular in the case of fault-tolerant algorithms. The interactions of the components are hard to trace and even trivial interdependencies may not be obvious when designing an algorithm on paper only. Hence, the analysis of new algorithms by means of computer aided verification and simulation becomes more and more state-of-the-art. Simulation in particular allows us to explore an algorithm's behavior in a very early phase of its design.

In addition to pure simulations, the model-checker approach allows us to use "wildcards" for which the tool is free to assign non-deterministic values. Hence, instead of a single simulation run that takes as input a specific test vector and analyzes the system behavior under this test, the model checker approach systematically searches the state space for all possible evaluations for each wildcard.

As we will discuss for the following two algorithms, simulation with SAL is used to add new functionality to the TTEthernet clock synchronization algorithm and to explore its behavior.

C. Formal Proof with SAL

As we gain more and more trust in the design of our algorithm we also have as a goal to actually formally prove some properties of interest. This is the true power of our formal framework, while we immediately deduce information through simulation, we can almost seamlessly switch to formal verification.

The proof of a property $\Box P$ ("P is always true") is done by k-induction [11], which is a generalized form of regular induction and consists of the following stages [12]:

- Base Case: Show that all the states reachable from I in no more than $k-1$ steps satisfy P
- Induction Step: For all trajectories $\sigma_0 \rightarrow \ldots \rightarrow \sigma_k$ of length k, show: $\sigma_0 \models P \wedge \ldots \wedge \sigma_{k-1} \models P \Rightarrow \sigma_k \models P$

k-induction is a powerful verification tool, but it can be directly applied only to relatively simple state-transition systems. For more complex systems we need to construct a system-level abstraction. Formally, a system-level abstraction $\mathcal{A}$ is also a state-transition system of the form $\langle \mathcal{S}, \mathcal{I}, \rightarrow_{\mathcal{A}} \rangle$, where $\mathcal{S}$ is a set of abstract states Σ_i and $\mathcal{I} \in \mathcal{S}$ is the initial abstract state. Furthermore, $\rightarrow_{\mathcal{A}}$ is a set of transitions between two abstract states. The system-level abstraction has to fulfill the following properties.

- Each system state σ_i is described by at least one abstract state Σ_j.
- The initial abstract state $\mathcal{I}$ describes at least one initial system state.
- For each transition in $\rightarrow$ that brings the system from a system state σ_1 to a system state σ_2 there either exists an abstract transition in $\rightarrow_{\mathcal{A}}$ such that σ_1 is described by the abstract state before the abstract transition and σ_2 is described by the abstract state after the abstract transition has been taken, or, if such an abstract transition does not exist, then the abstract state describing σ_1 must also describe σ_2.

Using the abstraction approach, the formal verification of a property $\Box P$ is done in two steps. In the first step we verify that the abstraction correctly represents the model $\mathcal{M}$ (i.e., it satisfies the properties listed above): $\mathcal{M} \models \mathcal{A}$ and in the second step we verify that the model $\mathcal{M}$ together with the abstraction $\mathcal{A}$ satisfies $\Box P$: $\mathcal{M} \wedge \mathcal{A} \models \Box P$.

We illustrate this abstraction method using the example of the verification of the TTEthernet clock synchronization algorithm as verified in [4]. In this model each SM is described by a state machine and all state machines are executed synchronously. For simplicity, we assume that each of these state machines has only two variables, `SM_state` and `SM_clock`, where `SM_state` is either `sync` or `send` and

`SM_clock` keeps track of the divergence from the perfect clock. The current system state is simply the sum of all of the current local states of the SMs.

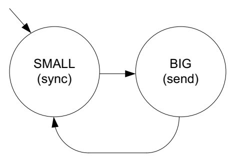

Figure 5: System-level abstraction for the formal proof

Figure 5 depicts a system-level abstraction for the TTEthernet that fulfills the abstraction properties listed above. In this case the abstraction is very simple and consists only of the two abstract states `SMALL` and `BIG`. `BIG` is an abstract state that requires all SMs to be in the `sync` state at the same time while in the abstract state `SMALL`, all SMs must be in the `send` state. Furthermore, *precision* will be bounded by some real constant `FACTOR_small` times $\max(\textit{drift})$ in the `SMALL` abstract state and by some other real constant `FACTOR` times $\max(\textit{drift})$ in the `BIG` state. `FACTOR_small` $<$ `FACTOR` holds and both numbers are derived manually or by re-running the model checking until no counterexamples are produced. These numbers depend on the number and type of failures present in the system and for TTEthernet `FACTOR` is between 2 and 4 (hence, the precision in TTEthernet is in $[2 \times \max(\textit{drift}), 4 \times \max(\textit{drift})]$.

It is easy to see that all SMs are consistently either in `sync` or `send`, given that all SMs start in the same state and change state synchronously. The resulting error of this synchronous approximation is covered in the term Δ_{error} of Equation 1. Note, as we already specify the precision in the abstraction, proving abstraction $\mathcal{A}$ makes the proof of the property $\Box P$ trivial. This results in high verification times for $\mathcal{A}$ and negligible ones for $\Box P$, as reported in [4] and in the following sections for the new algorithms.

We will discuss in the following two example algorithms how the basic TTEthernet clock synchronization model is updated with new system-level abstractions to derive automated and integrated formal proofs. We use a model of five SMs and two CMs as this is sufficient to show fault-tolerance for one and two faulty components. Although the statically configured number of components is a slight drawback of most model-checking approaches, the fast verification times and low memory utilization indicate that our verification approach also applies to a higher number of components.

IV. Diagnosis Algorithm

The TTEthernet clock synchronization algorithm is inherently fault-tolerant. However, the synchronization quality decreases with the number of faulty components and the severity of their failure modes. The diagnosis algorithm presented in this section aims to detect faulty TTEthernet devices, in particular faulty CMs, and remove them from the TTEthernet clock synchronization algorithm. By doing so, the failure mode of a faulty CM is transformed from an inconsistent-omission failure mode to a fail silent failure mode and we can formally verify that the diagnosis algorithm improves the precision in the system.

A. Algorithm Specification

The diagnosis algorithm is based on a simple accusation protocol presented by Algorithm 4 and Algorithm 5.

Algorithm 4 Diagnosis Algorithm executed by SM i

```
1: if $CM\_clock_j \neq \bot$ then
2:     $active[j] \leftarrow TRUE$
3: end if
4: for $j = 1 \rightarrow |CM|$ do
5:     if $CM\_clock_j = \bot \wedge active[j]$ then
6:         $accused[i][j], AC_i.accused[j] \leftarrow TRUE$
7:     end if
8: end for
```

Algorithm 4 is executed in the SMs immediately after the clocks are corrected (see Figure 2 on the temporal dependencies of the algorithms to each other). It starts with each SM recording those CMs from which they receive PCFs (lines $1-3$) using an array $active$ of boolean variables. The symbol $\bot$ denotes the absence of a PCF and in case that the clock of CM j is present (hence, the SM received a PCF from CM j) then the respective $active[j]$ will be set to $TRUE$.

In the remaining lines $(4-8)$ an SM i checks for each CM j whether it has been active before, but it did not receive a PCF in the current integration cycle. If this is the case, SM i accuses CM j to be omission faulty. For simplicity, we assume that this accusation information is stored in a local accusation matrix $accused[i][j]$, indexed by the SMs and CMs. Furthermore, SM i informs all other SMs of its accusation by sending and accusation message AC_i, where $AC_i.accused$ is a vector of boolean variables with each boolean representing a unique CM. SM i will set $AC_i.accused[j]$ if it accuses CM j. Again, for simplicity, we assume that the AC_i messages are sent as rate-constrained traffic on all redundant channels in a TTEthernet system. By the TTEthernet network it is, thus, ensured that the AC_i messages are delivered with a known upper bound in time and are transported over at least one non-faulty channel. We furthermore, assume that the exchange of the accusation information happens before the next execution of the TTEthernet clock synchronization algorithm.

Algorithm 5 is executed by an SM i that receives an accusation message AC_k from an SM k: when a boolean variable in $AC_k.accused[j]$ is $TRUE$, SM i sets the corresponding local $accused[k][j]$ to $TRUE$ as well. Each SM, thus, uses the matrix $accused$ to locally store all accusations from all SMs.

Algorithm 5 Accusation Message Reception in SM i

```
1: if $receives(AC_k)$ then
2:     for $j = 1 \rightarrow |CM|$ do
3:         if $AC_k.accused[j]$ then
4:             $accused[k][j] \leftarrow TRUE$
5:         end if
6:     end for
7: end if
```

Once an SM k accuses a CM or receives sufficient accusations for a CM, it will stop using the PCFs from this CM. Hence, the selection function presented in Algorithm 2 will also take the *accused* matrix into account when returning the set of CM_clock such that all CM_clock_j will be excluded that are accused by a sufficiently high number, z, of SMs. We can update line 7 in Algorithm 2 accordingly:

$$\begin{aligned}\{CM\_clock \quad | \quad & pcf\_memberhip\_new \geq accept\_threshold \\ & \wedge CM\_clock_j \Rightarrow \neg accused[k][j] \wedge \\ & CM\_clock_j \Rightarrow \neg \exists i_1, \ldots, i_z : \\ & accused[i_1][j] \wedge \ldots \wedge accused[i_z][j]\}\end{aligned}$$

An alternative realization to modifying the selection function is the deactivation of the communication link that connects the SM to the faulty CM.

Under a single failure fault-hypothesis, as assumed in this paper, $z = 2$ is sufficient and necessary to tolerate a faulty SM that may arbitrarily accuse CMs. In this case, an SM that receives two accusations for a CM can be certain that one of the accusations stems from a correct SM. Furthermore, as either an SM or a CM are faulty at the same point in time, a faulty SM excludes the presence of a faulty CM and, hence, even accusations of a faulty SM are distributed by all CMs consistently.

B. Simulation and Verification Procedure and Results

We have extended the basic model of the TTEthernet clock synchronization algorithm by the functionality of the diagnosis algorithm as presented in this section. Using our formal framework we can start by simulating the diagnosis and clock synchronization algorithms together. Figure 6 depicts such a simulation outcome of a scenario in a system of five SMs and two CMs where CM 1 is faulty in such a way that it may accept only a subset of SM_clock values. Hence, in general the compressed clocks, CM_clock_j, produced by the CMs will be different.

Figure 6 plots the divergence of the clock times from real time as described for Figure 4. In this scenario the clocks of SM 1-3 (denoted by Clock 1-3) have positive drift of 10 time units while the clocks of SMs 4 and 5 (denoted by Clock 4 and 5) have negative drift of 10 time units. In the first integration cycle all SMs receive PCFs from both CMs. In the second integration cycle all SMs except SM 2 receive PCFs from CM 1 (at x=3). Consequently, SM 2 will correct its clock slightly differently than the remaining SMs (at x=4). As SM 2 did not receive a PCF, it accuses CM 1 and will no longer accept PCFs from CM 1. As long as CM 1 does not fail to send a PCF to one of the other SMs, SM 2 always deviates from the remaining SMs after clock correction. However, in the fifth integration cycle (at x=9), CM 1 does not send a PCF to SM 3, which in turn also accuses CM 1. As now, SM 2 and SM 3 accuse CM 1 of being faulty, all SMs exclude CM_clock_1 from clock synchronization. Finally, from the sixth integration cycle on (at x=11) all SMs will only use CM_clock_2 for clock synchronization and the inconsistent omission failure mode of CM 1 is transformed into a fail-silence failure.

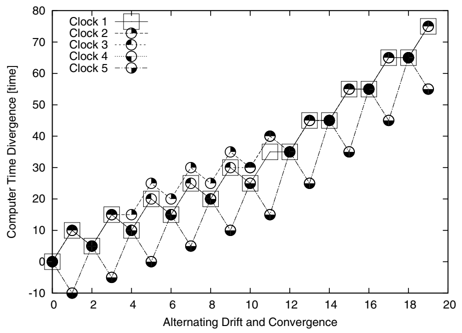

Figure 6: Example execution of the diagnosis algorithm as layered on top of the TTEthernet clock synchronization algorithm in presence of a faulty CM.

Simulation traces such as the one depicted above are valuable during the design of algorithms. However, our formal framework also allows us to formally verify the correctness of the diagnosis algorithm. For this we replace the system-level abstraction with a new one depicted in Figure 7. As shown, the abstraction extends the one of the TTEthernet clock synchronization algorithm with two abstract system states `SMALL_acc` and `BIG_acc`. This extension is a good example of how intuitively the abstraction process is: as long as the number of accusations is insufficient, the system transitions between `SMALL` and `BIG`. Once the accusations reach the threshold the `SMALL_acc` and `BIG_acc` are entered and the SMs transition between these two abstract states. The difference between the first two abstract states and the later ones is that the precision improves in `SMALL_acc` and `BIG_acc` (after a delay of one integration cycle).

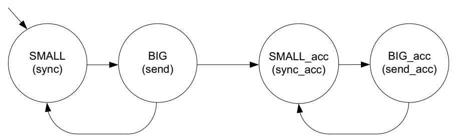

Figure 7: System-level abstraction for the formal proof

The diagnosis algorithm ensures that once the faulty CM is detected by a sufficiently high number of SMs, the exclusion of the faulty CM improves the precision from $\frac{8}{3} \times \max(drift)$ to $2 \times \max(drift)$.

Theorem 2.

$$\exists i_1, \ldots, i_z : accused[i_1][j] \wedge \ldots \wedge accused[i_z][j] \Rightarrow precision \leq 2 \times \max(drift)$$

Proof: Theorem 2 has been proven by model checking using `sal-inf-bmc`, k-induction and the abstraction as depicted in Figure 7 with the following performance characteristics (k represents the depth of the induction base and $time$ the verification time in seconds):

	k	$time$		k	$time$
Abstraction $\mathcal{A}$	5	180.00	Precision $\Box P$	4	3.55

■

C. Algorithm Discussion

The simple diagnosis algorithm can be extended in several ways some of which we discuss next. However, in this paper we do not consider this extensions in our analysis as our prime focus is on the demonstration of an integrated proof when layering a diagnosis service on top of an established clock synchronization algorithm rather than analyzing all variants of the diagnosis approach.

As a first extension, the SM may accuse a CM only after a configurable number of lost PCFs per configured time-interval. This would mitigate the probability that a CM is accused because of a transient error or because of a bit error as the Ethernet frame is transported over the communication link. Secondly, for the same reasons the accusation may be reset in all SMs to allow an accused CM to rejoin the TTEthernet clock synchronization algorithm. Lastly, the SMs may also take statistics on the number of lost application frames into account in their determination whether to accuse a CM or to remove an accusation.

V. Rate-Correction Algorithm

In this section we present the clock rate-correction algorithm that can be implemented as a layer on top of the TTEthernet clock synchronization algorithm. The rate-correction algorithm records the clock state-correction values for a configurable number of integration cycles. It then calculates an average of the corrected values and changes the rate of the clocks for a configurable percentage of this average. In any case the change of rate is bound by the maximum drift offset $\max(drift)$ from a perfect clock.

A. Algorithm Specification

Algorithm 6 is executed in each SM after the clocks have been corrected by the TTEthernet state-correction algorithm (Alg. 2, 3 in Figure 2). It consists of an observation phase (lines $1-3$) and the correction phase (lines $4-12$). In line 13 the integer variable $cycle$ is increased, which we use to count the integration cycles.

Algorithm 6 Rate-Correction Algorithm executed by SM i

```
1:  if $cycle \leq rate\_obs\_nr$ then
2:      $drift\_obs[cycle] \leftarrow act\_corr$
3:  end if
4:  if $cycle = rate\_obs\_nr$ then
5:      $corr \leftarrow \left(\sum_{l=1}^{l \leq rate\_obs\_nr} drift\_obs[l]\right) \div rate\_obs\_nr$
6:      if $corr > \max(drift)$ then
7:          $corr = \max(drift)$
8:      else if $corr < -\max(drift)$ then
9:          $corr = -\max(drift)$
10:     end if
11:     $clock\_rate \leftarrow clock\_rate - corr$
12: end if
13: $cycle \leftarrow cycle + 1$
```

The rate-correction algorithm starts with the observation phase in which the actual correction values that are calculated by the TTEthernet clock synchronization algorithm are stored for each integration cycle in the observation phase. To store, we use the array $drift\_obs[cycle]$ of real values. The observation phase completes after a configurable number of integration cycles $rate\_obs\_nr$ and the correction phase starts (line 4).

In the correction phase, the intermediate correction value that the algorithm first calculates is the arithmetic mean of the individual correction values (line 5). If the mean exceeds the configured maximum drift offset a non-faulty clock would exhibit (i.e., $\max(drift)$) the correction value $corr$ is reduced to these bounds (lines $6-10$). Finally, after the correction value is calculated and bounded it is used to correct the current rate of the local clock (line 11). Although it is not depicted in Algorithm 6, only a pre-configured percentage of the correction value may be used to correct a clock's rate.

Note that the modification of a clock's rate not necessarily demands a change of the physical oscillator frequency, but rather the number of oscillator ticks per integration cycle, which is configurable and can be changed. Hence, a change of a clock's rate is equivalent to increasing or decreasing the number of oscillator ticks per integration cycle.

B. Simulation and Verification Procedure and Results

In our formal analysis of the rate-correction algorithm we use a network of five SMs and two channels, where each channel implements exactly one CM. We have extended the formal model of the TTEthernet clock synchronization algorithm with the additional functionality described in Algorithm 6. Again, we start with some simulations to get confidence in the correctness of the design of the layered rate-correction algorithm as well as in the formal model of it. An example scenario is presented in Figure 8.

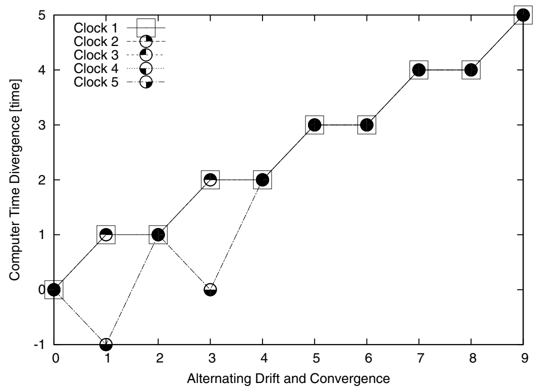

Figure 8: Fault-free scenario of the layered rate-correction algorithm with stable SM clock drifts

Figure 8 plots the divergence of the clock times from real time as introduced in Figure 4. Clocks 1 to 3 have a positive drift, while clocks 4 and 5 have a negative drift. The first two integration cycles are configured as the observation phase in which the nodes record their clock correction values. After the second integration cycle, the clocks calculate the *corr* value as specified in Algorithm 6 and adapt the rate of their clocks. In the scenario of Figure 8 *corr* does not exceed the $max(drift)$ and as depicted, from the third integration cycle onwards, all clocks are almost perfectly aligned.

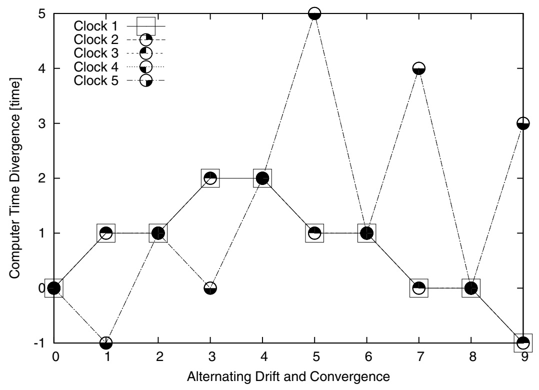

Figure 9: Fault-free scenario of the layered rate-correction algorithm with unstable SM clock drifts

The scenario as discussed above is certainly idealized as it reflects certainly strong assumption, as for example perfectly stable clock drifts. In reality, this will hardly be the case. Figure 9 shows a scenario with unstable clocks and resulting changing drift rates. Here, during the first integration cycle, clocks 1 to 3 have positive drift while clocks 4 and 5 have negative drift. Analogously to the stable drift scenario, clocks 4 and 5 are the only clocks that correct their clock state. At the end of integration cycle two clocks 4 and 5 have established $corr = 2$ (as they corrected $+2$ time units in each of the first two integration cycles). Now, the drift of the clocks changes, in a way that clocks 1 to 3 now drift in the negative direction while clocks 4 and 5 drift in the positive direction. Consequently, the correction value that clocks 4 and 5 apply adds up to the now positive drift and leads to an increase in the precision in the system.

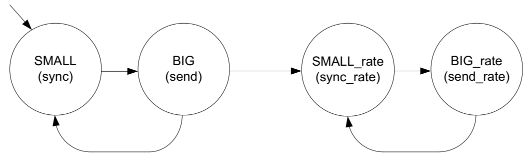

Figure 10: System-level abstraction for the formal proof

To formally verify properties about the rate-correction algorithm we define the system-level abstraction as depicted in Figure 10. The graph is essentially the same as for the diagnosis abstraction, however the underlying abstract states and transitions are, of course, different. The abstraction consists of four abstract states `SMALL`, `BIG`, `SMALL_rate`, and `BIG_rate`. `SMALL` and `BIG` represent the system during the observation phase, while `SMALL_rate` and `BIG_rate` represent the system when the clock rates are adapted. Again, the system-level abstraction very naturally reflects the algorithm phases.

We use this abstraction to verify the precision under the condition of unstable clock drifts. As discussed in the scenario of Figure 9 the precision may become larger when the drift of the clocks changes in the same direction as the rates are corrected to. The general theorem is as follows: the rate-correction algorithm ensures that, even under arbitrarily changing drift rates within the specified drift range, and in presence of an inconsistent omission faulty CM, the overall precision is bound by $8/3 \times 2 \times max(drift)$.

Theorem 3.

$$precision \leq \frac{8}{3} \times 2 \times max(drift)$$

Proof: Theorem 3 has been proven by model checking using `sal-inf-bmc`, k-induction and the abstraction as depicted in Figure 10 with the following performance characteristics (k represents the depth of the induction base and $time$ the verification time in seconds):

	k	$time$		k	$time$
Abstraction $\mathcal{A}$	5	1432.69	Precision $\Box P$	4	219.77

■

C. Algorithm Discussion

The rate-correction algorithm is a simple means to improve the precision in a system when the drift rates of the clocks can be assumed to be stable to some degree. However, even if they are not stable the rate-correction algorithm can improve the precision if the rate-correction algorithm is executed periodically and the change of the drift is relatively slow compared to the frequency of execution of the rate-correction algorithm.

As TTEthernet is intended as integrative network for mixed-criticality systems it may also be the case that some nodes of a network will be more affected by physical processes, like heat, than others. In this case, the system architect may configure more affected nodes as synchronization clients which only passively synchronize to the TTEthernet timeline as generated by the SMs and CMs. Even further, the system architect may decide to run the rate-correction algorithm on the synchronization clients more frequently than on the SMs.

The location of a node within the network can also influence the design decision on how often to run the clock-rate correction algorithm. Systems that are in spatial proximity to physical processes with varying temperature ranges, e.g., motor control, may have require to run the rate-correction algorithm frequently. Other systems may adjust their rate only after initial synchronization.

The detailed discussion of these architectural decisions is outside of the scope of this paper and we target at an evaluation in the context of a real-world application.

The rate-correction algorithm is executed in the SMs. Although, the CMs may adjust their clock rate to the SMs as well, the formal assessment of such configurations is outside of the scope of this paper and plan to explore this behavior in future work.

VI. Conclusion

This paper has introduced new distributed algorithms which can be implemented as layers on top of the TTEthernet clock synchronization protocol. We have presented a diagnosis algorithm and a clock rate-correction algorithm. The diagnosis algorithm follows a simple accusation protocol and aims to identify failure scenarios in which a faulty CM inconsistently distributes synchronization information. When such a faulty CM is diagnosed, then the non-faulty SMs consistently discard all synchronization information from the faulty CM. As a result of the diagnosis algorithm, the precision in the synchronized network improves and we have presented formal evidence for that. The clock rate-correction algorithm is executed in each of the SMs and continually records the clock state-correction values that the TTEthernet clock synchronization protocol calculates. After a configurable number of consecutive measurements all SMs use these measurements to adapt the rates of their clocks. By simulation we have shown, that the precision in the synchronized network improves and we have formally verified that the precision is still bounded when the clock drifts change arbitrarily within given bounds.

For our studies we have developed a novel simulation and verification framework that allows the integrated verification of algorithms such as the ones discussed above together with the underlying clock synchronization protocol. This formal framework is based on a model checker for infinite data types, which allows to realistically model real-time clocks. Furthermore, the framework enables push-button proofs that reduces the overhead of human deduction in the verification process to finding good system-level abstractions, which we have shown follows quite naturally from an algorithm's behavior.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement $n^{\circ}236701$ (*CoMMiCS*). The second author was supported by NASA Cooperative Agreement NNX08AC59A.

References

[1] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112 – 126, 2003.

[2] H. Kopetz, *TTP/C Protocol – Version 1.0*. Vienna, Austria: TTTech Computertechnik AG, Jul. 2002, Available at http://www.ttagroup.org.

[3] W. Steiner, *TTEthernet Specification*, TTA Group, 2008, Available at http://www.ttagroup.org.

[4] W. Steiner and B. Dutertre, "Automated formal verification of the ttethernet synchronization quality," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, Eds. Springer Berlin / Heidelberg, 2011, vol. 6617, pp. 375–390.

[5] W. Torres-Pomales, M. R. Malekpour, and P. Miner, *ROBUS-2: A Fault-Tolerant Broadcast Communication System*. Hampton, Virginia, USA: Langley Research Center, 2005.

[6] M. Serafini, P. Bokor, N. Suri, J. Vinter, A. Ademaj, W. Brandstätter, F. Tagliabo, and J. Koch, "Application-level diagnostic and membership protocols for generic time-triggered systems," *IEEE Trans. Dependable Sec. Comput.*, vol. 8, no. 2, pp. 177–193, 2011.

[7] *FlexRay Communications System - Protocol Specification - Version 2.1*. FlexRay Consortium, 2005, Available at http://www.flexray.com.

[8] H. Kopetz, A. Ademaj, and A. Hanzlik, "Combination of clock-state and clock-rate correction in fault-tolerant distributed systems," *Real-Time Systems*, vol. 33, no. 1-3, pp. 139–173, 2006.

[9] W. Steiner and B. Dutertre, "SMT-Based formal verification of a TTEthernet synchronization function," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, S. Kowalewski and M. Roveri, Eds., vol. 6371. Springer-Verlag, 2010, pp. 148–163.

[10] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "Tool presentation: SAL2," in *Computer-Aided Verification (CAV 2004)*, S. Verlag, Ed., 2004.

[11] L. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *Computer-Aided Verification, CAV 2003*, ser. Lecture Notes in Computer Science, A. Voronkov, Ed., vol. 2725. Springer-Verlag, 2003, pp. 14–26.

[12] B. Dutertre and M. Sorea, "Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata," in *Proc. of FORMATS/FTRTFT*, ser. Lecture Notes in Computer Science, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Springer-Verlag, Sep. 2004, pp. 199–214.