Automated Formal Verification of the *TTEthernet* Synchronization Quality

Wilfried Steiner[1] and Bruno Dutertre[2]

[1] TTTech Computertechnik AG, Chip IP Design
A-1040 Vienna, Austria
wilfried.steiner@tttech.com

[2] SRI International, Computer Science Laboratory
Menlo Park, CA 94025, USA
bruno@csl.sri.com

Abstract. Clock synchronization is the foundation of distributed real-time architectures such as the Timed-Triggered Architecture. Maintaining the local clocks synchronized is particularly important for fault tolerance, as it allows one to use simple and effective fault-tolerance algorithms that have been developed in the synchronous system model.

Clock synchronization algorithms have been extensively studied since the 1980s, and many fundamental results have been established. Traditionally, the correctness of a new clock synchronization algorithm is shown by reduction to these results. Until now, formal proofs of correctness all relied on interactive theorem provers such as PVS or Isabelle/HOL. In this paper, we present an automated proof of the *TTEthernet* clock-synchronization algorithm that is based on the SAL model checker.

1 Introduction

Distributed real-time systems are omnipresent in our daily lives and are becoming increasingly large and complex. It is becoming apparent that the correct development of such complex systems requires a sound architectural basis. The time-triggered architecture (TTA) [1] is intended to facilitate the development of fault-tolerant, real-time systems. TTA has been successfully adopted in industries that demand a high level of determinism, such as the avionics industry in which predictability of system operation is key. Upon others, *TTEthernet* (an implementation of the TTA) has been selected for the Orion Space Program [2]. The prime concept of TTA is a common perception of time in the devices that form the distributed system. These devices rely on local hardware clocks to build a common logical time base that is consistent across the system: any two logical clocks must read approximately equal values at any time during the system evolution. To maintain consistency, a clock synchronization algorithm must be used to compensate for the imperfection of the physical clocks. The maximal difference between two non-faulty logical clocks in the system is the *synchronization quality* or *precision* achieved by the algorithm.

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 375–390, 2011.

Clock synchronization has been studied for decades. Fundamental results provide answer to basic questions such as how well can clocks be synchronized in a distributed system [3] or how to construct fault-tolerant clock synchronization algorithms (e.g., [4]). Applications of these results to specific implementations and industrial products is described in several publications (e.g., [5]). Even with recent technological improvements in hardware clocks (e.g., embedding atomic clocks on a chip), clock synchronization remains highly relevant to modern real-time distributed systems. Fault-tolerant synchronization algorithms are required to align the clocks initially and to tolerate clock failures.

In many systems, safety depends critically on correct clock synchronization. As a consequence, significant effort has been dedicated to developing rigorous correctness proofs of various clock-synchronization algorithms. Schneider has shown that these algorithms share very similar properties and has introduced a general proof scheme for establishing their correctness [6]. Formal proofs of clock-synchronization algorithms have been developed by Rushby et al. [7], Shankar [8], and Miner [9] using the EHDM theorem prover; other formal proofs used PVS, the successor of EHDM [10,11]. Both EHDM and PVS are interactive theorem provers that require human guidance and expertise. Recently, more automated proof methods have been investigated that attempt to reduce the need for human expertise, by leveraging advances in model checking technology and automated reasoning engines known as SMT solvers. For example, Barsotti et al. [12] combine Isabelle/HOL and the SMT solvers CVC3 and Yices to formally verify Schneider's generic scheme. Another example of combined PVS and SAL proof method has been presented by Pike [13]. In [14], we used a model-checking approach to verify the "compression master" functionality of the *TTEthernet* clock synchronization algorithm (see next section). This verification was almost automated except that it required us to provide a few auxiliary lemmas by hand. In this paper, we extend the latter work to the full *TTEthernet* clock-synchronization algorithm and to analyzing the synchronization quality achieved by this algorithm. Model-checking clock synchronization algorithms has been done before in [15]. However, these studies are limited to four fully connected nodes and symbolic representation of time with fixed timing parameters. In this paper we treat time as a continuous entity while leaving the parameters uninterpreted. Thus, our proofs are valid for all timing parameterizations of the *TTEthernet* clock synchronization protocol.

This paper continues in the following section with an informal presentation of the *TTEthernet* clock synchronization algorithm. In Section 3 we then give an overview of the proof method and discuss the formal model in detail. We present the results of the formal proofs as well as some example testcases in Section 4. Finally, we conclude in Section 5.

2 *TTEthernet* Clock Synchronization Algorithm

TTEthernet is an extension of the traditional Ethernet standard, with additional services that guarantee reliable, deterministic delivery of time-critical messages.

A *TTEthernet* network consists of end systems and switches. End systems are connected to switches with bi-directional communication links and switches may connect to each other. 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 can implement redundant channels. An example network with two redundant channels is depicted in Figure 1.

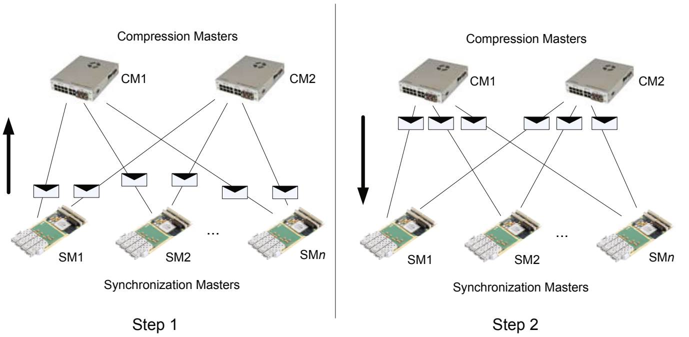


Fig. 1. Overview of the *TTEthernet* two step clock synchronization algorithm

2.1 Clock Synchronization Overview

End system and switches define physical components in the *TTEthernet* network and for the clock synchronization algorithm we use three different "roles": Synchronization Master (SM), Compression Master (CM), and the Synchronization Client (SC). For simplicity of discussion we assume a network consisting of five end systems and two channels, as depicted in Figure 1. Furthermore, end systems implement the SM role and the CMs are realized in the switches. SCs are only passively synchronizing to the timebase as maintained by the SMs and CMs and we exclude this role therefore from our discussion. In the clock synchronization algorithm SMs and CMs inform each other about their current state of their local clock by exchanging Protocol Control Frames (PCF).

In *TTEthernet* the clocks are synchronized in two steps. 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. The result of the convergence function is then delivered to the SMs in form of new PCFs (the "compressed" PCFs). In the second step the SMs collect the compressed PCFs from the CMs and execute a second convergence function. Our contribution in [14] has been restricted to showing the correctness of the implementation of the compression function, which we therefore assume in this paper.

TTEthernet requires 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 CMs implement a central guardian functionality that ensures that only one PCF per SM is used per re-synchronization cycle. Though, in the worst case, we assume that the clock value provided by a faulty SM can be arbitrary.

2.2 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. To minimize the impact of the faulty SMs *TTEthernet* uses a variant of the fault-tolerant median to calculate the new "compressed" clock. Following rules define the compressed clock depending on the number of SM_clock_i values received.

- one SM clock: $compressed\_clock = SM\_clock_1$
- two SM clocks: $compressed\_clock = \frac{SM\_clock_1 + SM\_clock_2}{2}$
- three SM clocks: $compressed\_clock = SM\_clock_2$
- four SM clocks: $compressed\_clock = \frac{SM\_clock_2 + SM\_clock_3}{2}$
- five SM clocks: $compressed\_clock = SM\_clock_3$
- more than five SM clocks: average of the $(k+1)^{th}$ largest and $(k+1)^{th}$ smallest clocks, where k is the number of faulty SMs to be tolerated.

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. In addition to the compressed clock value, the CMs also generate a membership vector *pcf_membership_new*. Each position in this vector is assigned to one and only one SM. The CMs will set the bit of a SM, if the respective SM i has provided a local clock value SM_clock_i and will clear the bit otherwise. The CMs transmit the *pcf_membership_new* vector in the payload of the compressed PCF. The self-checking pair design of the CM guarantees that the compressed clock and the *pcf_membership_new* vector are consistent. Hence, the design prevents a faulty CM to set an arbitrary number of bits in *pcf_membership_new*.

2.3 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 sorted in increasing order. Under the assumption of one CM per channel and up to three channels maximum, the convergence function has to cover following three cases:

- one CM clock: $SM\_clock = CM\_clock_1$
- two CM clocks: $SM\_clock = \frac{CM\_clock_1 + CM\_clock_2}{2}$
- three CM clocks: $SM\_clock = CM\_clock_2$

In the case of a faulty CM, a SM may receive at maximum one compressed PCF per CM (as the faulty CM may decide not to send its compressed PCF to some SMs). Furthermore, a SM will only use a compressed PCF in the convergence function discussed above if the *pcf_membership_new* field has at least *accept_threshold* of bits set. *accept_threshold* is calculated as follows:

1. *current_max* = maximum of bits set in the *pcf_membership_new* field of any compressed PCF
2. *accept_threshold* = *current_max* minus the allowed number of faulty SMs

The SM will discard a compressed PCF that has less than *accept_threshold* bits set in the *pcf_membership_new* field. This mechanism ensures that a SM excludes compressed PCFs that represent relative 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.

2.4 Clock Synchronization Example

Figure 2 gives and example scenario of the *TTEthernet* clock synchronization. The x-axis represents progress in time as alternating intervals of clock drift and re-synchronization using the two-steps approach. Note that these are logic steps and do not represent real time. Odd values on the x-axis represent the SM local clock values immediately before the synchronization, even values represent the values of the SM local clocks immediately after synchronization. The y-axis depicts the clock-time of the SMs. We will discuss the representation of the clock time in the next section.

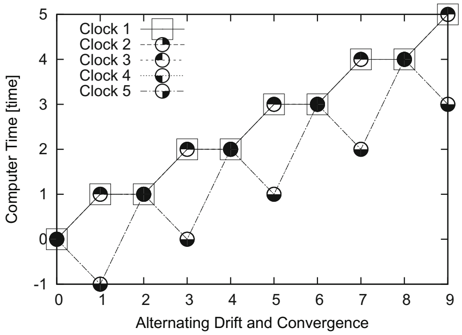


Fig. 2. Fault-free scenario of the *TTEthernet* clock synchronization algorithm

The example depicted in Figure 2 shows a fault-free execution trace of five SMs and two CMs. Initially, the SMs are perfectly synchronized. SMs 1, 2, and 3 have maximum positive drift and SMs 4, 5 have maximum negative drift. As there are no failures involved and when neglecting digitalization errors and transmission jitter on the network, the local clocks of the SMs become perfectly re-synchronized with each execution of the clock synchronization algorithm.

3 Automated Formal Verification Procedure

We give an overview of the proof method next. We then discuss the formal model in the SAL notation and the proof procedure.

3.1 Proof Method Overview

The *TTEthernet* clock synchronization algorithm has been formalized in SAL [16] 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 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 continuous entity and can use k-induction [17] as proof method. The proof of a property $\Box P$ by k-induction is a generalized form of regular induction and consists of following stages [18]:

- 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 that $\sigma_0 \models P \wedge \ldots \wedge \sigma_{k-1} \models P \Rightarrow \sigma_k \models P$

In our studies we have observed an interesting dependency between k and the synchronization quality: increasing k allowed to calculate the upper bound on the precision more tightly. This means there is a trade-off between the depth (k) of the proof and the quality of its result (calculated upper bound on the precision).

The SMs are modelled as state machines with two states representing the alternating drift and correction intervals. The example scenario in Figure 2 also gives an overview of our modelling method. As we are only interested in the maximum difference between any two non-faulty local clocks, we can abstract from the nominative length of the synchronization interval. All we need to model is the maximum difference to the nominative length that would result from a non-faulty clock. In many current industrial use cases the drift offset, i.e., the offset as a result of the imperfect physical local clocks, is the dominant part of this offset and we refer therefore to the offset as “drift offset”. Note, although we use the term drift offset we implicitly also take into account network jitter, digitalization errors and similar error terms. We argue that these effects can be summarized by a sufficiently high value for what we call the drift offset. As we do not specify a particular value for the drift offset in our proofs, but only require an upper bound on it, the proofs are also valid for real systems rather than only for idealized models. We have been able to directly proof the value of the precision in certain *TTEthernet* networks by only specifying the functionality of the SM and the CM without any additional lemma or further modelling tricks. However, we see a significant performance gain if we use a lemma informing the model-checker that all SMs consistently change their state (from the drift interval to the correction interval and vice versa). For this lemma we use a simple system level abstraction as introduced in [18].

3.2 Formal Model

```
POSREAL: TYPE = {x: REAL | x>=0 };
max_drift: POSREAL; max_clock: REAL; max_SM:  NATURAL = 5; max_CM: NATURAL = 2;
```

The formal model[1] starts with some constants and types. `POSREAL` defines the positive real numbers. `max_drift` describes the absolute value of the maximum drift offset of a clock within one re-synchronization interval. `max_clock` describes the time horizon. Both, `max_drift` and `max_clock`, have no value assigned, hence, we leave them “uninterpreted”. This means that they may have any value. `max_SM` defines the maximum number of SMs in the network. `max_CM` defines the number of redundant channels in the network. We define exactly one CM per channel.

We define dedicated types to denote the sets of nodes, SMs, channels etc. The formal model is then executed fully synchronously in alternating steps `send` and `sync` as denoted by the SM’s state.

```
TYPE_drift: TYPE = REAL; TYPE_clock: TYPE = REAL;
TYPE_SM: TYPE = [1..max_SM]; TYPE_CM: TYPE = [1..max_CM];
TYPE_states: TYPE = {send, sync};
```

In the `send` state the SMs provide the values of their local clocks to the CMs which execute the first step convergence function and return the converged values back to the SMs. In the `sync` state, the SMs execute the second step convergence function and update their local clock accordingly. We discuss this process in more detail next based in the SM and CM implementation in SAL.

3.2.1 Synchronization Master Module. The synchronization master module `SM` is parameterized by `TYPE_SM`, to identify a particular SM by `id`.

[1] A more detailed report and the models can be found at `http://sal-wiki.csl.sri.com`

```
SM[id:TYPE_SM]: MODULE = BEGIN
INPUT list_compressed_clock: ARRAY TYPE_CM OF TYPE_clock
OUTPUT state: TYPE_states, clock: ARRAY TYPE_CM OF TYPE_clock
LOCAL drift: TYPE_drift, interval_ctr: NATURAL
```

The SMs receive their input from the CMs. list_compressed_clock represents the first-step converged clock values, i.e., the compressed clock value. An SM will output its current state and the value of its local clock clock. clock is modeled as an array of size TYPE_CM, which allows us to model inconsistent faulty behavior of a faulty SM as discussed later on. In addition to the input and output we also define some local variables in for an SM. drift defines the drift offset for a given re-synchronization interval. interval_ctr counts the re-synchronization intervals; it is used to derive test traces. We initialize the model to a clean state.

```
INITIALIZATION  interval_ctr = 0; state = sync; clock = [[j:TYPE_CM] 0];
                drift IN {x: TYPE_drift | x=-max_drift OR x=max_drift};
```

We use the formal model for both testcase generation and formal proof of the synchronization quality. Depending on the purpose of the formal experiment drift can be set to a static value to pretty-print counterexamples or to an arbitrary value. In the case above, our aim is to generate a nice trace for which we initialize drift to take either the positive or the negative maximum drift offset. The model checker is free to chose either value once for the complete execution of the model. The SAL construct IN models this non-deterministic choice. It is interpreted as: let drift be an x which satisfies the condition as specified above. We use the IN construct at several positions in our model.

In case of the formal proof we want cover a more general case of clock drift, for which we have to define drift as a DEFINITION.

```
DEFINITION  %drift IN {x: TYPE_drift | x>=-max_drift AND x<=max_drift};
```

The “%” sign indicates a comment line in SAL. We use it here to emphasize that drift may either be initialized or defined, but not both. The definition of drift says that in every step of the model execution drift may take an arbitrary value in between the maximum negative and positive drift offset and this value may change with each step. We use this definition for the formal proofs.

```
[  state=sync -->
   state'=send;  interval_ctr'=interval_ctr + 1; clock'=[[j:TYPE_CM] clock[j] + drift];
[] state=send -->
    state'=sync;
    clock' IN {x: ARRAY TYPE_CM OF TYPE_clock |
      x[1]=x[2] AND average(list_compressed_clock[1], list_compressed_clock[2],x[1])}; ]
```

In the fault-free case there are only two transitions in the the state machine of an SM (expressed by guarded commands in the form guard --> commands). When the SMs are in the sync state their local clocks are closely synchronized. The next state will be send for which they increase the counter of the re-synchronization intervals, and select a new value for their local clocks. This new value is simply the sum of the current clock value and the drift offset as specified by `drift`. Our treatment of `clock` is different from the traditional correctness proofs which aim to show that clock-time simulates real-time with a certain accuracy. This is not necessary in our approach. We are only interested in the maximum difference of any two `clock` values of non-faulty components. Hence, we update `clock` only for the differences in the nominative length of the re-synchronization interval and can omit its actual length. In the `send` state the local clocks of the SMs are far apart and they process the compressed clock values received from the CMs to bring the local clocks back into agreement for the following `sync` state.

In a faulty-free system with two channels and one CM per channel, the SM applies the arithmetic average to the received compressed clock values. In the transition of the SM we specify that `clock` shall take a new value such that the `average` predicate is satisfied. The predicate is satisfied when the third parameter is the arithmetic mean of the first two parameters.

```
average(value1, value2, avg: TYPE_clock): BOOLEAN = avg=(value1+value2)/2
```

3.2.2 Compression Master Module.

The CM is parameterized by `TYPE_CM`, such that `id` identifies a particular CM. It takes the clock values as input and returns the compressed clock value to the SMs as a result of the first step convergence. The CM uses the local variable `order` to sort the clock values as provided by the SMs.

```
CM[id:TYPE_CM]: MODULE = BEGIN
INPUT clocks_cm: ARRAY TYPE_SM OF TYPE_clock
OUTPUT compressed_clock: TYPE_clock
LOCAL order: ARRAY TYPE_SM OF TYPE_SM
```

We model the CM as a stateless process. Its only purpose is the calculation of the first step convergence function, the compression function. In a system with even number of SMs or more than five SMs the CM has to apply the averaging function as discussed previously.

```
compressed_clock IN {x: TYPE_clock | average(clocks_cm[order[2]], clocks_cm[order[3]], x)}
```

In a system with one, three, or five SMs, the compressed clock is simple the middle value, e.g., for five SMs (`compressed_clock=clocks_cm[order[3]];`). In both cases `order` determines the order of the clock values. We have introduced this method in [14] and summarize it here for completeness. We define `order` to be an array of SM identifiers that satisfies the `sort` predicate. `sort` is satisfied when `sorted_list` is an array in which the entries point to the elements of `unsorted_list` in increasing order.

```
order IN {x: ARRAY TYPE_SM OF TYPE_SM | sort(clocks_cm, x)};
sort(unsorted_list: ARRAY TYPE_SM OF TYPE_clock,
     sorted_list:ARRAY TYPE_SM OF TYPE_SM): BOOLEAN =
(FORALL (i:TYPE_SM): i<max_core =>
      unsorted_list[sorted_list[i]] <= unsorted_list[sorted_list[i+1]]) AND
(FORALL (i,j:TYPE_SM): sorted_list[i] = sorted_list[j] => i=j);
```

3.3 Automated Formal Proof

We are interested in verifying the precision in the system (property `distance`):

```
distance: LEMMA world |- G(FORALL (i,j: TYPE_SM):
 (list_states[i]=send AND list_clocks[i][1]>list_clocks[j][1] =>
  list_clocks[i][1]-list_clocks[j][1]) <= FACTOR*max_drift));
```

`distance` says that when the SMs are in the `send` state, which is just before the execution of the clock synchronization algorithm, the maximum difference of any two local clocks is bound by `FACTOR*max_drift` (when faulty SMs are present they have to be excluded). As introduced earlier, `max_drift` is the maximum drift offset of a correct clock in the system from real time within one synchronization inverval. The value of `FACTOR` has to be assigned by hand in the model. This value is typically the result of an informal analysis of the algorithm. E.g., in the case of faulty CM we suspect that the value is (8/3). When `FACTOR` has not been determined upfront, we can even "search" for it by manually testing assignments for `FACTOR` until the model checker stops producing counter-examples and proves `distance` to be correct.

We invoke `sal-inf-bmc` using the following command, where `clocksync` is the model name, `--depth 3` specifies the analysis depth, and `-i` invokes k-induction:

```
> sal-inf-bmc clocksync distance --depth=3 -i
```

This direct proof works well for a low number of nodes and relatively benign failure modes. We can speed-up the verification time significantly with a simple abstraction method introduced in [18]. For this we define two abstract system states, `BIG` and `SMALL`.

```
abstractor: MODULE =
BIG = (FORALL (i:TYPE_SM): list_states[i]=send) AND
      (FORALL (i,j:TYPE_SM): list_clocks[i][1]>list_clocks[j][1] =>
           (list_clocks[i][1]-list_clocks[j][1] <= FACTOR * max_drift));
SMALL = (( ... <= FACTOR_small * max_drift));
```

In the system-level abstraction we formulate that all SMs are at the same time either in the `send` state or in the `sync` state and they are synchronously proceeding between the two states. Furthermore, we already define `FACTOR*max_drift` here, which makes the proof of `distance` later on trivial. We can proof that the system-level abstraction `abstract_invar` is correct and verify `distance` while using `abstract_invar` as lemma (option `-l`).

```
> sal-inf-bmc clocksync abstract_invar --depth=3 -i
> sal-inf-bmc clocksync distance -l abstract_invar --depth=3 -i
```

4 Fault-Injection Experiments and Results

In this section we show how to model failures and discuss the *TTEthernet* synchronization quality. We use a system model consisting of five SMs and two CMs. We want to show the synchronization quality of the *TTEthernet* clock synchronization algorithm under a single SM failure, a single CM failure, and under concurrent SM and CM failures.

4.1 Inconsistent Omission Faulty CM

In *TTEthernet* the failure mode of a CM is inconsistent omission faulty. Hence, a faulty CM may arbitrarily decide which clock values from the SMs to use and which to discard. It can also arbitrarily decide to which SMs it will send it's compressed clock value. However, even a faulty CM will correctly represent the set of SMs that is selected in the *pcf_membership_new* vector in its compressed PCF. We define CM with id 1 to be the faulty CM (`FAULTY_CM: NATURAL=1`).

A non-faulty CM receives clock values from all SMs. As in our network five SMs are present and they all are non-faulty (for now), a correct CM receives five SM values. According to the algorithm definition (Sec. 2.2) it selects the median value, i.e., the third value as its compressed clock value.

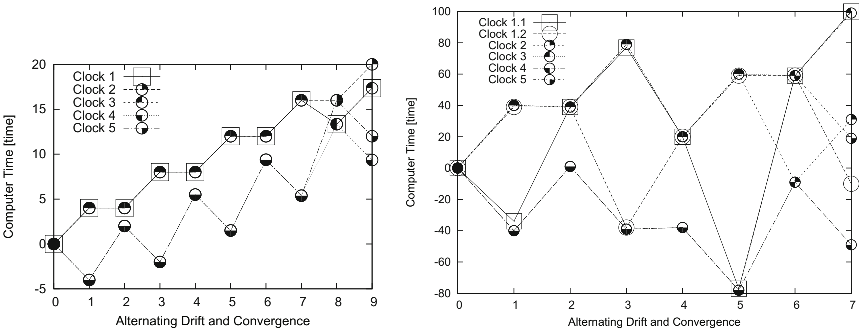
```
compressed_clock IN {x: TYPE_clock |
  IF id /= FAULTY_CM THEN x=clocks_cm[order[3]]
  ELSE x=clocks_cm[order[3]] OR
   average(clocks_cm[order[2]], clocks_cm[order[3]], x) OR
   average(clocks_cm[order[2]], clocks_cm[order[4]], x) OR
   average(clocks_cm[order[3]], clocks_cm[order[4]], x)
  ENDIF};
```

The inconsistent-omission faulty CM may accept only an arbitrary subset of the five SM clock values, but this choice is reflected in the number of bits it sets in the *pcf_membership_new* vector. As the correct CM will deliver its compressed PCF with a *pcf_membership_new* vector having five bits set, the *accept_threshold* will be four. Hence, in order that there is a chance at all that the compressed clock of the faulty CM is not excluded in the second convergence step in the SMs it may only discard one of the SMs clock values. There are four options for the CM to calculate the first step convergence function, as depicted in the SAL model above. In the first case the CM accepts all SM clocks, and the following three cases cover when it discards any one of the SM clocks.

In order to simulate the inconsistent-omission faulty transmission behavior of the faulty CM we define a second transition in the SM state machine for the `send` state.

```
[] state=send
    --> state'=sync; clock' = [[j:TYPE_CM] list_compressed_clock[CORRECT_CM]];
```

Hence, we map the inconsistent transmission failure of the CM to an non-deterministic choice in the SM: the SM is free to decide whether it received a clock value from the faulty CM or not.



(a) (b)

Fig. 3. Example scenarios of the *TTEthernet* clock synchronization algorithm with a faulty CM (left), and a faulty SM and CM (right)

Figure 3(a) gives an example trace of the algorithm execution in presence of a faulty CM. Again, the x-axis represents alternating intervals of drift and convergence and the y-axis the clock time. In contrast to Figure 2 we see the impact of the faulty CM resulting in a non-zero difference in the local clocks of the SMs after re-synchronization. We have formally verified the precision in this system setup to be (8/3) × *drift_offset*, by k-induction at depth three. `FACTOR` = (8/3) has been calculated from an informal reasoning, which is also depicted in Figure 3(a): some SMs have fast clocks, some slow ones, and the faulty CM sends its compressed clock to only one of these groups. Consequently, only the SMs in one group correct their clocks towards the respective other one. In the figure we see that the faulty CM provides it clock only to the SMs with negative drift, which correct their clocks, while the SMs with positive drift do not correct their clock as they only receive the compressed clock values from the correct CM.

4.2 Arbitrarily Faulty SM

An arbitrarily faulty SM is free to fake its local clock values. We define the SM with id 1 to be faulty (`FAULTY_SM: NATURAL=1`). The communication of the local clock values is modelled by an array indexed by the CMs and the faulty SM may send different clock values to different CMs by assigning different values to different array entries. We model the arbitrary clock value by the failure term `failure` that simulates the faulty local clock values. `failure` can take any value for the faulty SM and is 0 for non-faulty SMs. Finally, we update the transition in the SM to reflect the failure (i.e., a change in the update of `clock'`).

```
LOCAL failure: ARRAY TYPE_CM OF TYPE_drift
failure IN {x: ARRAY TYPE_CM OF TYPE_drift |
     IF id = FAULTY_SM THEN TRUE ELSE x[1]=0 AND x[2]=0 ENDIF};
clock'= [[j:TYPE_CM] clock[j] + drift + failure[j]];
```

The impact of a faulty SM only on the precision of the network is limited, even non-existent. Although the CMs can receive different clock values from the faulty SM and consequently derive different compressed clocks, they still will send their compressed clock values to all SMs. Hence, all non-faulty SMs receive the compressed clocks from the CMs consistently. The precision of the system with an arbitrarily faulty SM is, thus, the same as the precision in a fault-free system: 2 × *drift_offset*.

4.3 Inconsistent Omission Faulty SM and CM

One particular failure combination of interest is when SM and CM are inconsistent-omission faulty. Hence, both may accept only a subset of clock values and send their clock value to only a subset of SMs or CMs. We can reuse the modelling of the faulty CM and have to introduce two additional transitions for the faulty SM. First, a faulty SM may decide to receive only the compressed clock from the faulty CM, and, secondly, the faulty SM may decide not to receive any compressed clock.

```
[] state=send AND id=FAULTY_SM
 --> state'=sync; clock' = [[j:TYPE_CM] list_compressed_clock[FAULTY_CM]];
[] state=send AND id=FAULTY_SM
 --> state'=sync;
```

For completeness, we note here that the remaining case of the faulty SM receiving only the compressed clock of the correct CM has been already covered by the transition below by modelling the faulty CM.

```
[] state=send
 --> state'=sync; clock' = [[j:TYPE_CM] list_compressed_clock[CORRECT_CM]];
```

In addition to the model of the SM we also have to add additional cases to the calculation of the compressed clock in the CM. This can be done in a systematic way as depicted in the SAL source code below. There are two general cases, in the first case the faulty SM provides a clock value to the correct CM. This case is identical with the behavior of the faulty CM scenario discussed previously. In the second case the faulty SM does not provide a clock value to the correct CM. For this case we define the predicate `order_part` which is identical to the `order` predicate, except that it only orders the clock values from the correct SMs. The correct CM will calculate the compressed clock as the average from the second and third clock value. The faulty CM is free to use either option as discussed previously. In addition it may use the average as the correct CM or it may even decide to accept only three of the four correct SM clocks. In the latter case compressed clock from the faulty CM is either the second or the third correct SM clock.

```
compressed_clock IN {x: TYPE_clock |
  ( ... case as faulty CM only ... ) OR
(IF id /= FAULTY_CM THEN average(clocks_cm[order_part[2]], clocks_cm[order_part[3]],x)
 ELSE ( ... case as faulty CM only ... )  OR
      average(clocks_cm[order_part[2]], clocks_cm[order_part[3]],x) OR
      x=clocks_cm[order_part[2]] OR x=clocks_cm[order_part[3]] ENDIF) }
```

For a *TTEthernet* network with five SMs and two CMs and an inconsistent-omission faulty SM and CM we have verified the precision to be bound by 4 × *drift_offset*.

4.4 Inconsistent Omission Faulty CM and Arbitrarily Faulty SM

The failure modelling of a network with inconsistent-omission fault CM and arbitrarily faulty SM is simply the combination of the individual failure models as introduced above. Figure 3(b) shows an example trace of the failure scenario. The main difference to the previous figures is that the clock of the faulty SM 1 is depicted as two clocks 1.1 and 1.2. This is, again, because the faulty SM may send different values to the different CMs, and, indeed, this scenario is shown in Figure 3(b). On the odd numbers on the x-axis the local clock readings just before the algorithm execution are depicted. Some clocks are fast and some are slow, and the faulty SM supports both groups. We also see that arbitrarily faulty behavior of the clock of SM 1, as the jumps from one extreme to the other. We have proven the precision to be (12/3) × *drift_offset* in this configuration of five SMs and two CMs with faulty CM and SM. This number also confirms an informal argument of the worst-case scenario similar to the one discussed for a faulty CM only.

4.5 Summary of Verification Results

The verification times are summarized in Table 1. The precision Π for the scenarios follows from `FACTOR` as Π = `FACTOR` × *drift_offset*. “distance” gives the verification times without system level abstraction. “abstraction” shows the verification times of the invariant for the system level abstraction, i.e., the verification that the abstraction is correct, and the last row depicts the verification times of “distance” when the abstraction is used as lemma.

Table 1. Verification results; `FACTOR` is a scalar, verification times are given in seconds

Property	No Faults	Faulty			
		CM	SM	CM/SM io	CM/SM a
`FACTOR`	2	(8/3)	2	(12/3)	(12/3)
distance	10.5	28.25	8.66	N/A	N/A
abstraction	0.5	0.58	0.49	85.3	44.43
distance+abst.	0.34	0.36	0.38	0.39	0.4

We clearly observe that the verification times decrease dramatically when the system-level abstraction is used. For difficult failure scenarios it is even essential to derive a formal proof. The arbitrarily faulty SM and inconsistent omission faulty CM faulty scenario terminates at depth five after eight-hundred seconds without counterexample and without proof. The inconsistent omission CM SM scenario returns the same result after sixteen-hundred seconds (indicated by N/A). Note that “N/A” in the table means that the inductive proof without the abstraction lemma was not possible. However, "distance" has been proven by k-induction using the abstraction lemma (as shown in the last row of the table), providing full coverage of our failure assumptions.

5 Conclusion

In this paper we have shown for the first time that fault-tolerant clock synchronization proofs can be fully automatized even in a model of continuous uninterpreted time. This is a significant advancement over the state-of-the-art which involves heavy-duty theorem provers or imposes significant modeling restrictions. We have shown that the precision in a *TTEthernet* network is between two and four times the drift offset (including network jitter and digitalization effects), depending on the failures to be tolerated. The only step requiring human interaction that one may argue being required in the synchronization verification is in the definition of the failure cases to model faulty components realistically. However, as we have discussed, the failure model can be constructed fairly systematically. For *TTEthernet* the failure cases are limited by design. For more complex protocols it can make sense to separately model check for the completeness of these cases. In our experiments we used a system of five Synchronization Masters and two Compression Masters. While this is a small system, industry trends indicate that mostly a core set of nodes for clock synchronization is used anyhow. Hence, the limitation to a small number of clocks does not impose an industrial shortcoming. On the other hand, given the fast verification times and low memory use of our approach, we will target larger systems in future work.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement $n°236701$ (*CoMMiCS*). The second author was supported by NASA Cooperative Agreement NNX08AC59A. The authors would like to thank Günther Bauer for the informal proofs on the precision and feedback to this paper.

References

1. Kopetz, H., Bauer, G.: The Time-Triggered Architecture. Proceedings of the IEEE 91(1), 112–126 (2003)
2. Howard, C.E.: Orion avionics employ COTS technologies. In: Avionics Intelligence (June 2009)
3. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control 62(2-3), 190–204 (1984)
4. Lamport, L., Melliar-Smith, P.M.: Byzantine clock synchronization. In: PODC 1984: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, pp. 68–74. ACM, New York (1984)

5. Kopetz, H.: TTP/C Protocol – Version 1.0. Vienna, Austria: TTTech Computertechnik AG (July 2002), http://www.ttagroup.org
6. Schneider, F.B.: Understanding protocols for byzantine clock synchronization. Cornell University, Ithaca, NY, USA, Tech. Rep. TR87–859 (1987)
7. Rushby, J., von Henke, F.: Formal verification of the interactive convergence clock synchronization algorithm. Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. SRI-CSL-89-3R, (February 1989), http://www.csl.sri.com/papers/csl-89-3/ (revised online August 1991)
8. Shankar, N.: Mechanical verification of a generalized protocol for byzantine fault-tolerant clock synchronization. In: Vytopil, J. (ed.) FTRTFT 1992. LNCS, vol. 571, pp. 217–236. Springer, Heidelberg (1992)
9. Miner, P.S.: Verification of fault-tolerant clock synchronization systems. NASA, NASA Technical Paper 2249 (1993), http://ntrs.nasa.gov
10. Schwier, D., von Henke, F.: Mechanical verification of clock synchronization algorithms. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 262–271. Springer, Heidelberg (1998)
11. Pfeifer, H., Schwier, D., von Henke, F.: Formal verification for time-triggered clock synchronization. In: Weinstock, C.B., Rushby, J. (eds.) Dependable Computing for Critical Applications, vol. 7, pp. 206–226 (January 1999)
12. Barsotti, D., Nieto, L., Tiu, A.: Verification of clock synchronization algorithms: experiments on a combination of deductive tools. Formal Aspects of Computing 19, 321–341 (2007)
13. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: Proceedings of Formal Methods in Computer Aided Design (FMCAD 2007), pp. 231–238. IEEE, Los Alamitos (2007)
14. Steiner, W., Dutertre, B.: SMT-Based formal verification of a TTEthernet synchronization function. In: Kowalewski, S., Roveri, M. (eds.) FMICS 2010. LNCS, vol. 6371, pp. 148–163. Springer, Heidelberg (2010)
15. Malekpour, M.R.: Model checking a byzantine-fault-tolerant self-stabilizing protocol for distributed clock synchronization systems. NASA, Tech. Rep. NASA/TM-2007-215083 (2007)
16. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: Tool presentation: SAL2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114. Springer, Heidelberg (2004)
17. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Voronkov, A. (ed.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)
18. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 199–214. Springer, Heidelberg (2004)