SMT-Based Formal Verification of a *TTEthernet* Synchronization Function

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. *TTEthernet* is a communication infrastructure for mixed-criticality systems that integrates dataflow from applications with different criticality levels on a single network. For applications of highest criticality, *TTEthernet* provides a synchronization strategy that tolerates multiple failures. The resulting fault-tolerant timebase can then be used for time-triggered communication to ensure temporal partitioning on the shared network.

In this paper, we present the formal verification of the compression function which is a core element of the clock synchronization service of *TTEthernet*. The compression function is located in the *TTEthernet* switches: it collects clock readings from the end systems, performs a fault-tolerant median calculation, and feedbacks the result to the end systems. While traditionally the formal proof of these types of algorithms is done by theorem proving, we successfully use the model checker `sal-inf-bmc` incorporating the `YICES` SMT solver. This approach improves the automatized verification process and, thus, reduces the manual verification overhead.

1 Introduction

Modern networked systems host a multitude of applications often with varying criticality levels. In an on-board network of an airplane, for example, highly critical flight-management and control applications are implemented as well as less critical video applications. To ensure independence between these applications, traditionally a federated network approach is realized in which different applications use private networks. However, with the increasing number of applications the federated approach becomes costly and, as a consequence, there is a tendency throughout many industries to converge from a multitude of heterogeneous federated networks to an integrated communication infrastructure.

TTEthernet (Time-Triggered Ethernet [1,2]) is such a communication infrastructure for mixed-criticality systems. For traffic of highest criticality, *TTEthernet* provides time-triggered communication. Time-triggered communication, also

S. Kowalewski and M. Roveri (Eds.): FMICS 2010, LNCS 6371, pp. 148–163, 2010.

known as time-division multiple-access (TDMA), is a communication paradigm in which the local clocks of the communication participants are synchronized, and frames are dispatched and relayed according a communication schedule defined *a priori*. Hence, as the local clocks in the participants are synchronized, the communication schedule is executed synchronously and contentions at the network are avoided. Time-triggered communication provides therefore strong temporal partitioning because the possibility that two or more communication participants access the network at the same point in time can be excluded by design and enforced by simple guardian mechanisms. The synchronized local clocks are the fundamental prerequisite for time-triggered communication, and the correctness of the synchronization algorithms is therefore essential.

The main contribution of this paper is the discussion and formal verification of the compression function which is a core element of the *TTEthernet* fault-tolerant synchronization strategy. We present the verification of several properties of different characteristics (membership and clock synchronization) and discuss their different computational overhead.

The subject of clock synchronization is very well understood, with a broad academic foundation developed as early as in the nineteen-eighties (e.g. [3], [4]). Our work proves the correctness of a particular implementation of these fundamental results. Still, there is also a certain novelty in the compression function: the compression function runs unsynchronized to the synchronized timebase; its core functionality is the collection of local views of the global synchronized timebase and the generation of a consolidated new reference point. The approach presented in this paper can easily be applied to enhance master-slave based clock synchronization systems to multi-master systems, in which the compression function operates as proxy for fault-tolerant clock synchronization.

Formal proofs of this kind of algorithms have been traditionally done by theorem proving [5], [6]. In this paper we discuss the application of the SMT-based verification approach introduced in [7] and [8] to fault-tolerant clock synchronization problems. To our knowledge this is the first time that model-checking has been applied to the verification of a convergence function such as the fault-tolerant median.

The formal models are free for download from the SAL wiki[1] to foster cooperation in the current ongoing standardization process of *TTEthernet* (SAE AS6802) as well as for upcoming inter-operability and conformance tests. While this paper discusses the *TTEthernet* low-level synchronization functions, the higher-level synchronization strategy focusing on startup/restart is presented in [9]. Because of space limitations, we present and discuss only parts of the formal model. The full model and a more detailed verification report are described in [9].

This paper is structured as follows: we give an overview of *TTEthernet* and an informal description of the compression function in the next section. Section 3 provides an overview of the formal model. We present the verification procedure and results in Section 4. Finally, we conclude in Section 5.

[1] `http://sal-wiki.csl.sri.com`

2 *TTEthernet* Informal Discussion

2.1 Communication of Synchronization Information

Figure 1 depicts an example *TTEthernet* network consisting of five end systems and two redundant communication channels. Channel 1 consists of three switches, where one of the switches is configured as *Compression Master* (CM) and the other switches are configured as *Synchronization Client*s (SC). Channel 2 consists of a single switch configured as CM. All end systems are configured as *Synchronization Master*s (SM). The synchronization procedure is initiated by the SMs which send synchronization messages, called *Protocol Control Frame*s (PCF), to the CMs. The CMs process the proposed PCFs and relay new PCFs back to the SMs and SCs. SCs will relay the PCFs from the SMs to the CMs and vice versa but use only the PCFs from the CMs for their own synchronization.

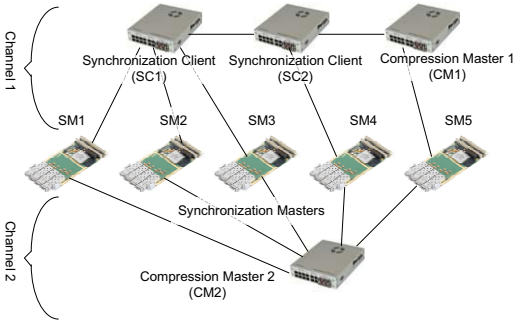

Fig. 1. Example TTEthernet network

TTEthernet implements a so called "permanence function" that compensates for network jitter of PCFs: as a PCF flows through the network, all devices that relay the frame add their delay imposed on the PCF into a dedicated field of the frame. Hence, a receiver can determine the actual latency of a PCF through the network with negligible error. The permanence function is then a simple method executed in the receiver to transform network jitter into network latency: (a) we calculate offline the maximum network latency considering all PCFs; (b) upon reception of a PCF the receiver artificially delays the PCF for the remaining difference between this maximum network latency and the actual latency as transported in the PCF. Hence, the "transmission" of each PCF will always take the maximum network latency.

To highlight the difference between the point in time of physical reception and the point in time when the frame is actually used in the CM, we use the term "*permanence point in time*" for the latter (see Figure 2). The permanence function allows us to abstract from network jitter and to treat the network latency as a constant. Without loss of generality we assume a zero network latency in the formal proofs: at the point in time when a PCF is dispatched

by a SM it is immediately "*permanent*" at the CM. The negligible error of the permanence functions are covered by the modelling of the clock drift. The automatized formal proof of the permanence function using `sal-inf-bmc` can be found in [9].

2.2 Compression Function Informal Description

During synchronized operation mode, the SMs dispatch their PCFs at the same nominal point in time to the CMs. Due to drifts in the oscillators, the actual dispatch points in the SMs and the resulting permanence points in time in the CMs will deviate. Therefore, the CMs implement a so called "*compression function*" that runs unsynchronized to the synchronized global time. The compression function collects the PCFs from different SMs and produces a new PCF which is sent back to the SMs. The dispatch point in time of this new PCF is calculated as a function of the relative permanence points in time of the PCFs from the SMs. This dispatch point in time from the CM is called the "*compressed point in time*". The focus of this paper is to verify the correct relation between the permanence points in time and the compressed point in time.

The compression function runs unsynchronized to the synchronized timebase. It is started upon the reception of a PCF, rather than upon the synchronized local clock in the CM reaching a particular point in time. Therefore, it has to be guaranteed that faulty SMs that may send early or late will not cause the compression function to recognize only a subset of PCFs from correct SMs in the generation of the new PCF.

Figure 2 depicts an example execution of the compression function. In this example three end systems that are configured as SMs dispatch PCFs, in particular a special type called Integration Frame (IN), to a switch that is configured as CM. The depicted deviation of the dispatch points in time stem from the relative differences in the oscillators of the end systems; in a perfect world, these dispatch points in time would be perfectly aligned.

CM will use the permanence function discussed previously to derive the permanence points in time of the PCFs. The first permanence point in time (p_1) will cause the compression function to start the collection phase. As successive PCFs become permanent, the CM records their offsets relative to the first permanence point in time ($p_i - p_1, i > 1$) and stores these offsets in a local data structure that we call the *clock synchronization stack*. The duration of the collection phase is given by the following rules, where "observation window" specifies the maximum deviation of two correct local clocks in the system as measurable by a clock within the network:

- The first permanence point in time will cause the compression function to collect the following permanent PCFs for one observation window.
- When the compression function collects at least a second permanent PCF during the first observation window, the collection phase is prolonged for a second observation window.

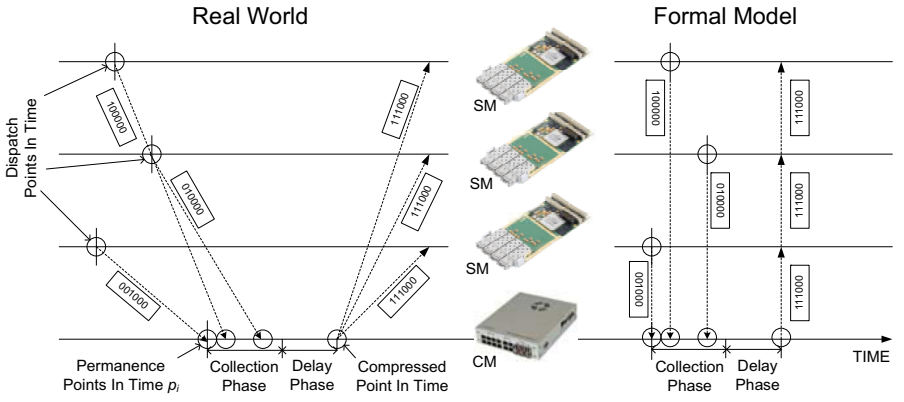

Fig. 2. Compression function overview, three end systems configured as SM provide their local clock readings to a switch configured as CM. In the real world the network jitter is compensated by the permanence function.

- The collection phase will end when the number of permanent PCF collected during observation window i is equal to the number of permanent PCFs collected during observation window $i-1$ (hence, when no new PCF became permanent for the duration of one observation window). Otherwise collection will be continued for another observation window.
- The collection phase will stop at the latest after the $(k+1)^{th}$ observation window, where k is the configured number of faulty SMs to be tolerated.

After the collection phase the relative permanence points in time of the collected PCFs are used to determine a correction value for the following delay phase. In order to minimize the impact of the faulty SMs we use a variant of the fault-tolerant median (where $p_i, i \geq 1$ represent the permanence points in time):

- 1 permanence point in time: $correction\_value = 0$
- 2 permanence points in time: $correction\_value = \frac{p_2 - p_1}{2}$
- 3 permanence points in time: $correction\_value = p_2 - p_1$
- 4 permanence points in time: $correction\_value = \frac{((p_2 - p_1) + (p_3 - p_1))}{2}$
- 5 permanence points in time: $correction\_value = p_3 - p_1$
- more than 5 permanence points in time: take the average of the $(k+1)^{th}$ largest and $(k+1)^{th}$ smallest inputs, where k is the number of faulty SMs that have to be tolerated.

In the delay phase, the compression function will wait for

$$\begin{aligned} delay = {} & correction\_value + \\ & (k+1) * observation\_window - \\ & collection\_phase\_duration \end{aligned}$$

where *collection_phase_duration* is the length of the preceding collection phase.

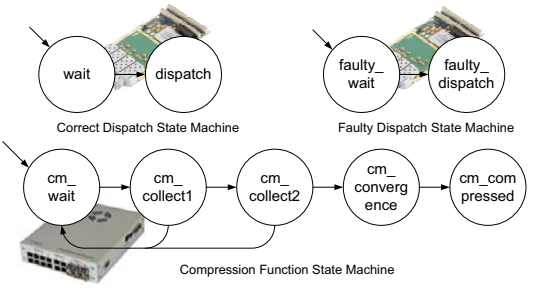

Fig. 3. State machines for the faulty and the correct dispatch processes as well as for the compression function

Figure 3 depicts the state machines of the dispatch processes in the SMs and the state machine for the compression function in the CMs.

The dispatch process is described by a very simple state machine consisting of only two states: `wait` and `dispatch` (or `faulty_wait` and `faulty_dispatch` for faulty components). The dispatch process maintains a local timer variable that identifies the dispatch point in time. When this dispatch point in time is reached the dispatch process dispatches the PCF and enters the `dispatch` state (or `faulty_dispatch`). Once in `dispatch` or `faulty_dispatch`, a dispatch process will remain in that state forever.

The state machine of the compression function consists of the following states: `cm_wait`, `cm_collect1`, `cm_collect2`, `cm_convergence`, and `cm_compressed`. The compression function starts in the `cm_wait` state and enters the `cm_collect1` state when the first PCF becomes permanent. The `cm_collect1` and `cm_collect2` represent the collection phase and `cm_convergence` represents the delay phase of the compression function as described above. Finally, the compression function enters the `cm_compressed` state.

We are interested in verifying the correctness of the collection phase as well as the delay phase in the compression function, which results in the following four properties:

- `agreement`: when the compression function collects one permanence point in time of a PCF sent from a correct SM it will also collect permanence points in time from all other correct SMs within the same collection phase.
- `window`: the compressed point in time will be within the interval $[k * observation\_window, (k+2) * observation\_window]$.
- `correction`: all correct SMs will perceive the compressed point in time not more than *observation_window* from when they expect the compressed point in time.
- `termination`: the compression function process will produce a result.

3 Formal Model

Figure 4 gives an overview of the formal model used to verify the compression function. It consists of N modules that represent a dispatch process and the

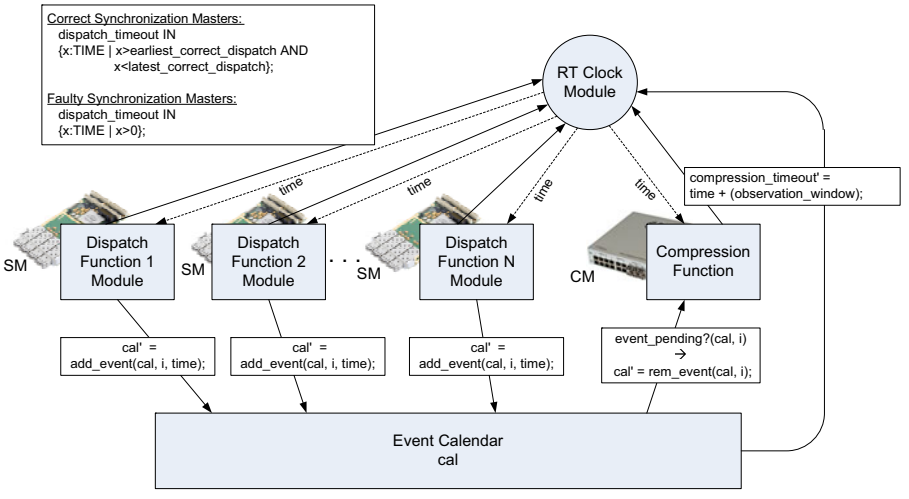

Fig. 4. Compression function model overview

compression function. An additional RT (Real-Time) Clock Module manages the timing in the model: the dispatch and compression modules receive the current time as an input from the RT Clock module. Furthermore, these modules each maintain a timeout output reflecting the next point in time when the module will produce its next event. The RT Clock module receives all timeout outputs as input and advances time to the smallest timeout value.

Clock drifts resulting in deviations of the local clocks and therefore different dispatch points in time are modelled by uncertainty intervals, which means that the end systems will set their new timeout to a non-deterministic position within parameterizable bounds.

The global data structure "Event Calendar `cal`" is used to model the flow of a PCF from the dispatch processes to the compression function. The transmission of a PCF by a SM is done by adding an event to the event calendar; likewise the reception of the PCF by the CM is modelled by the consumption of this event.

More detail on the progress of time using calendars as well as the exchange of messages can be found in [7].

The faulty SMs are simulated by allowing their dispatch points in time to occur at an arbitrary instant, while a correct SM dispatches its PCF within the specified uncertainty interval.

In the following we describe some parts of the model in the SAL notation. It starts with some general definitions.

```
k: NATURAL = 3;
N: NATURAL = 3*k + 1;
OBSERVATION_WINDOW_ID : TYPE = [1..k+1];
```

`k` defines the number of faulty SMs that have to be tolerated and `N`, the number of overall SMs required to tolerate the defined number of failures, is then given

by N=3*k+1. The SMs are represented by the dispatch functions described by the state machines above. DISPATCH_ID identifies the SMs in the system by numbering them [1..N]. Similarly, OBSERVATION_WINDOW_ID labels the observation windows from [1..k+1].

```
observation_window: REAL = 5;
earliest_correct_dispatch: REAL= (k+1)*observation_window;
latest_correct_dispatch: REAL = earliest_correct_dispatch + observation_window;
end_of_time: REAL = latest_correct_dispatch + ((k+1)+2)*observation_window;
```

Besides the number of faulty SMs to be tolerated, the length of the observation window is the only other parameter that has to be assigned by hand. All other parameters in the system are derived from those two. In this setup we set observation_window=5. As observation_window is the only temporal parameter that we assign a particular value, it does not matter what this value is: 5 represents $5\mu sec$ as well as $5sec$ or any $\frac{x*5}{y} sec, x, y > 0$. The earliest_correct_dispatch and the latest_correct_dispatch define the uncertainty interval when a correct SM dispatches its PCF. The definition of this interval contributes to the hypothetical worst case, in which the faulty SMs would send their PCFs in such a way that the collection phase in the compression function (which lasts $k + 1$ observation windows at most) could complete without collecting any PCF stemming from a correct SM. By definition, all correct SMs will dispatch their PCF within an interval of length observation_window. end_of_time is used to initialize the timeout variable of the reactive modules. The compression function is the reactive module that initially waits for the reception of PCFs. In order to prevent the compression function module from blocking the progress of time, we initially set its value to the point in time when execution of the compression function would be finished in the worst case.

Real-time is modelled analogously to [7] using a dedicated real-time clock module. For the compression function we need additional data structures and functions in order to collect the permanence point in times of PCFs and to calculate their fault-tolerant median.

```
clock_readings: TYPE =
 [# valid: ARRAY DISPATCH_ID OF BOOLEAN, value: ARRAY DISPATCH_ID OF TIME #];
```

clock_reading defines the clock synchronization stack, the data-structure that we use for storing the relative permanence points in time of PCFs that the compression function receives during its collection phase. empty_clock_readings defines the empty clock synchronization stack.

```
add_clock_reading(cr: clock_readings, i: DISPATCH_ID, v: TIME): clock_readings =
                              ((cr WITH .valid[i] := TRUE) WITH .value[i] := v);
```

add_clock_reading specifies a function for collecting values in the clock readings data-structure. The values are added in a stack-like fashion, so the relation between clock reading entry to SM will be lost. Whenever a new value is added, valid is set to TRUE and the value field holds the relative difference to the first permanence point in time p_1. The fault-tolerant median calculation is specified according to the requirements given in the informal discussion.

```
ft_median(cr: clock_readings): TIME = % for k=2, N=7
 IF    cr.valid[7] THEN (cr.value[3] + cr.value[5])/2
 ...
 ELSIF cr.valid[4] THEN (cr.value[2] + cr.value[3])/2
 ...
 ELSE cr.value[1]  ENDIF;
```

The algorithms are modelled as guarded commands of following form:

```
guard --> list of commands
```

The correct SMs dispatch their PCF within the uncertainty interval; faulty SMs may dispatch their PCF at any time. An example guarded command for a correct SM is given below.

```
dispatch_state = wait AND dispatch_timeout = time
-->
dispatch_state' = dispatch; dispatch_pit' = time;
cal'            = add_event(cal, i, time); dispatch_timeout' = time+end_of_time;
```

When SM is in `wait` state and the RT Module signals that time has reached its dispatch event, the SM will dispatch its PCF by adding an event to the calendar. Furthermore, it locally stores the current point in time, which we use in the formal proof, and sets its timeout output to a high value such that it does not block time progress.

We describe some core transitions of the compression function module next. The first transition describes the reception of the first PCF, which starts the collection phase.

```
[([] (i:DISPATCH_ID):
   event_pending?(cal, i) AND event_time(cal, i) = time AND compression_state = cm_wait
-->
compression_state' = cm_collect1;
compression_timeout' = time + (observation_window);
reading_index' = 2; last_reading_index' = 2;
membership_new'=[[index:DISPATCH_ID] IF index=i THEN TRUE ELSE membership_new[index] ENDIF];
pit_0' = time;
clock_stack'=add_clock_reading(clock_stack,reading_index,0);
cal' = rem_event(cal, i);)
```

When the compression function is in the `cm_wait` state and a new entry is added to the calendar, the transition to `cm_collect1` state is triggered. Note that we abstract from the transmission delays that would naturally occur in the *TTEthernet* network. We justified this abstraction in Section 2.

`reading_index` is used both for counting the number of permanent PCFs and as an index in the clock synchronization stack, where it points to the next free entry. `last_reading_index` is used to store the number of permanent PCFs collected until the latest observation window has been started. When the collection phase is started, the `reading_index` and the `last_reading_index` are updated and the entry in the `membership_new` bitvector for the SM that triggered the transition is set. `pit_0` is used to store the current point in time when the transition is triggered (which is p_1). Note, that in a real implementation this timestamp would be taken from an internal clock in the CM, rather then the current point

in real-time, which naturally is not present in any component. However, as we do not use `pit_0` directly, but only relative offsets to it, we conclude that our modelling does not introduce invalid additional information. Finally, 0 is added as the first entry to the clock synchronization stack and the entry to the calendar that triggered the transition is removed from the calendar.

The next transition is triggered at the end of an observation window i ($i \geq 2$).

```
[] compression_state = cm_collect2 AND time = compression_timeout
 AND reading_index >  last_reading_index AND window_counter < k+1
 -->
 compression_state' =  cm_collect2; compression_timeout' = time + observation_window;
 last_reading_index' = reading_index;
 window_counter' = IF window_counter=N THEN window_counter ELSE window_counter+1 ENDIF;
```

In this transition we check whether the number of permanence points in time has increased during the last observation window. If so, and it was not the last observation window yet, we continue the collection for another observation window. `window_counter` is used to keep track of the number of observation windows.

The next transition is taken when the number of permanence points in time is equal to the number collected during the previous collection window (hence, no new PCF has become permanent during the latest observation window), and at least $k + 1$ PCFs have been received. The state machine proceeds then to the `cm_convergence` state. The duration of the delay phase is calculated based on the relative permanence points in time, and the timeout is set accordingly to simulate the delay phase.

```
 compression_state=cm_collect2 AND reading_index=last_reading_index
 AND time=compression_timeout AND window_counter<k+1 AND reading_index>k+1 %proof only
 -->
compression_state' =  cm_convergence; window_counter' = window_counter;
compression_timeout'= time+ft_median(clock_stack)+(k+1-window_counter)*observation_window;
```

The compression function will stay in the `cm_convergence` state for the duration of the delay value.

```
 compression_state = cm_convergence AND time = compression_timeout
-->
 compression_state' = cm_compressed; compressed_true' = TRUE;
```

When real-time indicates the timeout of the delay value, the compression function transitions to the `cm_compressed` state and sets `compressed_true` to TRUE.

Once in the `cm_compressed` state, the compression function will stay in this state forever, setting the `compressed_true` flag to FALSE immediately after entering. Hence, `compressed_true` marks exactly one instant in real-time, which is used as the reference for clock correction in the higher-layer synchronization protocol. This instant marks the compressed point in time (`cm_compressed_pit`).

For the proof of termination of the compression function we define two transitions:

```
[] compression_state=cm_wait AND time=compression_timeout --> compression_state'=cm_error;
[] compression_state=cm_error --> compression_state' = cm_error;
```

The first transition says that, when the compression function is in `cm_wait` state for too long it will enter the dedicated error state `cm_error`. The second transition is there to avoid a deadlock in the error state.

4 Verification Procedure and Results

The proof of the compression function builds on the abstraction method introduced in [7]. In our assessment, we extend this approach to allow a configurable number of faulty dispatch processes. Furthermore, we add a dedicated error state that is entered when the compression function is not finished in time. This allows us to also verify a termination property of the compression function. For the abstraction, we first define abstract system states and the abstract transitions between them: the composition of the SMs and the CM results in the product automata of their respective state machines. An abstract system state is a subset of states in the product automata and the abstract transitions are between these subsets. We prove the correctness of the abstraction (lemma *abstract_inv*), which is then used in the verification of our properties of interest. The proofs are done by k-induction.

`sal-inf-bmc` provides assistance in the construction of the abstraction via counterexamples. Given that we defined an abstraction consisting out of two abstract states $A1, A2$ and an abstract transition from $A1$ to $A2$ a typical counterexample during the design phase could be as follows: SAL shows how a transition in one of the original state machines, say in an SM, imposes an abstract transition from $A1$ to an abstract state $A2'$ other than $A2$ which may be undefined yet. Resolving this situation can be done by either restricting $A1$, extending $A2$, or introducing a new $A2'$ with the respective abstract transition.

4.1 Abstraction Description

The system abstraction is depicted in Figure 5.
A 1: This is the initial abstract state when all dispatch functions and the compression function have assigned their local variables as well as the global calendar to the initial values.
A 2: In this abstract state, at least one of the dispatch functions has dispatched a PCF modelled by adding the respective entry in the calendar.

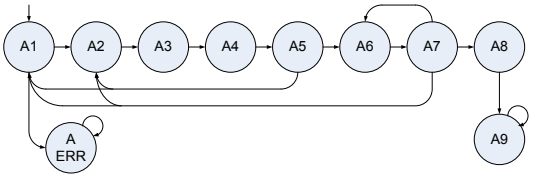

Fig. 5. System-Level Abstraction

A 3: In the A 3 abstract state, the compression function has consumed at least one of the permanent PCFs from the calendar and has started the first observation window of the collection phase.
A 4: This is the abstract state representing the first observation window of the collection phase of the compression function.
A 5: In the A 5 abstract state the collection phase has completed the first observation window. In this state we check whether to continue the collection of values or to restart the compression function.
A 6: This abstract state, again, represents the collection phase throughout one particular observation window for observation window $i \in 2..(k+1)$.
A 7: The A 7 abstract state is used to check at the end of each observation window $i \geq 2$, whether more PCFs have become permanent during the latest observation window $i-1$ and whether the collection phase operated already for $(k+1)$ observation windows. If the number of permanent PCFs has increased and the number of observation windows collected so far is below $k+1$ then the abstract state A 6 is entered again. If the number of permanent PCFs has not increased and the number of permanent PCFs is smaller than $k+1$, then the compression function is restarted. If the number of permanent PCFs has not increased, but the number of permanent PCFs so far is higher than $k+1$ then the abstract state A 8 is entered. Also, when the collection phase has reached the end of the $(k+1)^{th}$ observation window the abstract state A 8 is entered.
A 8: In this abstract state the compression function waits for the duration of the delay value calculated from values on the clock synchronization stack and the duration of the collection phase.
A 9: This is the final abstract state.
A ERR: This is the error state entered, when the compression function fails to terminate within a given timeout.

4.2 Key Disjunctive Invariants and Related Functions

The key in verification of the agreement and timing properties is in relating the individual states of the SMs to the state in the CM. For the agreement property this relation is a simple count of those SMs that have dispatched their PCF to the counter used in the CM. For the timing properties the relation is more complex as we not only have to formulate the relation based on the number, but also on the sequence in which the SMs dispatched their PCF.

4.2.1 Invariant for the Agreement Property. The agreement property can be verified using an invariant that describes the equality: the *number* of SMs that have dispatched their PCF is equal to the counter in the CM (`reading_index`).

```
reading_index =
IF count_msg(1, cal, list_dispatch_states, list_dispatch_pits,
             pit_0, window_counter, old_values)< N
THEN count_msg(1, cal, list_dispatch_states, list_dispatch_pits,
               pit_0, window_counter, old_values)+ 1
ELSE N ENDIF
```

The count_msg function counts those dispatch functions idx that have already dispatched their PCF (list_dispatch_states[idx] = dispatch) and which have been consumed by the compression function (NOT cal.signal[idx]).

4.2.2 Invariant for the Timing Properties. The verification of the window and correction property is more challenging than for the agreement property. Here we not only have to relate the number of SMs to the CM state, but the *sequence* in which the SMs dispatched their PCF. Analogously to the reading_index used above, the CM uses the clock synchronization stack clock_stack to locally store the relative differences of the frame permanence points in time. On the other hand we store the individual dispatch points in time in the list_dispatch_pits[i], where i is the index of the SMs.

For the invariant we now have to define how the clock_stack relates to the list_dispatch_pits[i]:

```
(FORALL (i:DISPATCH_ID):
IF    i = 1 THEN clock_stack.value[i]=0
ELSIF i <= count_memb(1, membership_new)
  THEN  clock_stack.value[i] =  list_dispatch_pits[observed_order[i]]
                              - list_dispatch_pits[observed_order[1]]
ELSE clock_stack.value[i]=0 ENDIF)
```

We know that the first entry on the clock synchronization stack will always be 0. Furthermore, the number of values on the clock synchronization stack is determined by the number of PCFs received by the SM so far. This number can be obtained from the membership vector membership_new, using a simple count function (count_memb). The $\texttt{i}^{th}$ value on the clock synchronization stack will be the temporal distance between the $\texttt{i}^{th}$ PCF and the first PCF that has become permanent.

To determine the first and the $\texttt{i}^{th}$ PCF requires some type of sort procedure on list_dispatch_pits[i]. As it turns out, this is a little tricky in our formalism as an explicit sort algorithm works only for a very small number of values. To overcome this limitation we use a declarative approach: we introduce observed_order as a new array and observed_order[i] shall be assigned the index of the SM that provided the $\texttt{i}^{th}$ PCF. Hence, observed_order is not the sorted version of list_dispatch_pits, but rather a sorted array of pointers to list_dispatch_pits. In SAL this can be done via a non-deterministic selection (using the IN construct) and a predicate:

```
observed_order IN {x: ARRAY DISPATCH_ID OF DISPATCH_ID | sort([[i:DISPATCH_ID]
   IF NOT membership_new[i] THEN time+1 ELSE list_dispatch_pits[i] ENDIF], x)};
```

Here we say, that observed_order is some array x spanning over the SMs, which satisfies the sort predicate. The sort predicate simply takes a modified version of list_dispatch_pits and x as input. Note that the modification of list_dispatch_pits is necessary to exclude PCFs that have become permanent in a collection phase prior to the latest one.

```
sort(unsorted_list: ARRAY DISPATCH_ID OF TIME,
     sorted_list: ARRAY DISPATCH_ID OF DISPATCH_ID): BOOLEAN =
  (FORALL (i:DISPATCH_ID): i<N =>
     unsorted_list[sorted_list[i]] <= unsorted_list[sorted_list[i+1]]) AND
  (FORALL (i,j:DISPATCH_ID): sorted_list[i]=sorted_list[j] => i=j);
```

Finally, `sort` returns true when its second parameter is an ordered pointer list.

4.3 Verification Properties and Results

```
agreement: LEMMA system |- G(compression_state=cm_compressed =>
                              (FORALL (i:DISPATCH_ID): i<=k OR membership_new[i]));
```

`agreement` says that once the compression function has reached the `cm_compressed` state, all correct SMs are present in the `membership_new` vector.

```
window: LEMMA system |- G(compression_state=cm_compressed AND compressed_true =>
 (FORALL (i:DISPATCH_ID): i<=k OR
  (list_dispatch_pits[i]+k*observation_window<=time_out[COMPRESSION_FUNCTION_ID] AND
   time_out[COMPRESSION_FUNCTION_ID]<=list_dispatch_pits[i]+(k+2)*observation_window)));
```

`window` says that the `cm_compressed_pit` occurs in a window of size `2*observation_window`.

```
correction: LEMMA system |- G(compression_state=cm_compressed AND compressed_true =>
(FORALL (i:DISPATCH_ID): i<=k OR
(time_out[COMPRESSION_FUNCTION_ID] - list_dispatch_pits[i] + (k+1)*observation_window
    <= observation_window) OR
(list_dispatch_pits[i]+ (k+1)*observation_window-time_out[COMPRESSION_FUNCTION_ID]
    <= observation_window)));
```

In a perfect world all `sm_dispatched_pit` would occur at the same point in time resulting in a nominal `cm_compressed_pit` of `(k+1)*observation_window` later. `correction` says that all SMs will observe the actual `cm_compressed_pit` with a maximum deviation of one `observation_window` from the nominal `cm_compressed_pit`. Hence, all correct SMs will have to correct their local clocks for a maximum of one `observation_window`.

Note that the `window` and `correction` properties do not account for the network latency and jitter (as these are abstracted by the permanence function). Hence in the real world the nominal `cm_compressed_pit` will occur *max_transmission_delay* later than reflected in the properties above.

```
termination: LEMMA system |- G(compression_state/=cm_error);
```

The `termination` property says that the `cm_error` state will never be reached. Hence, `termination` ensures that eventually the `cm_compressed` state is reached and trivial solutions to the previous properties are excluded.

The results of our model-checking assessment are presented in Table 1, where N is the number of SMs of which k are faulty. For each scenario we also give the number of SMT variables and SMT assertions.

Table 1. Verification results for the compression function properties

Property	k	N	Verif. Time	#var	#assert	k	N	Verif. Time	#var	#assert
agreement	1	4	1.65 sec	633	606	2	7	2.58 sec	964	955
window	1	4	1.81 sec	633	720	2	7	7.49 sec	964	1136
correction	1	4	1.80 sec	633	721	2	7	4.07 sec	964	1036
termination	1	4	1.72 sec	633	723	2	7	2.67 sec	964	1134
abstract_inv	1	4	6.51 sec	633	727	2	7	2,227.38 sec	964	1036

As depicted, the main computation time is consumed in the verification of the abstract invariant, while the verification time of the actual properties is small. Verification runs for $k = 3$ have been aborted after several hours. Although, the approach is not scalable for high k, it is sufficient for the verification of dual fault-tolerance as required in the original *TTEthernet* specification. The main reason for this computational complexity is the non-deterministic selection construct used in the definition of the `observed_order` array, as this results in a quadratic number of SMT constraints. Table 2, shows the verification results for a restricted compression function model that only models the membership part.

Table 2. Verification results for membership only

Property	k	N	Verif. Time	#var	#assert	k	N	Verif. Time	#var	#assert
agreement	1	4	0.97 sec	509	454	2	7	1.51 sec	766	663
agreement	3	10	2.02 sec	1023	872	4	13	2.51 sec	1280	1081
agreement	5	16	3.13 sec	1537	1290	6	19	3.57 sec	1794	1499
abstract_inv	1	4	1.27 sec	509	490	2	7	2.57 sec	766	840
abstract_inv	3	10	6.92 sec	1023	1113	4	13	75.35 sec	1280	1386
abstract_inv	5	16	508.57 sec	1537	1659	6	19	10,056.97 sec	1794	1932

Again, we see that the main computational complexity is in the verification of the abstract invariant. Indeed, the membership-only verification allows us to increase the system size quite significantly from seven to nineteen SMs (with $k = 6$).

5 Conclusion

In this paper, we discussed the formal verification of the *TTEthernet* compression function, which is essential for its application in safety-critical and mixed-criticality systems.

We have shown how `sal-inf-bmc` can be applied to the formal verification of fault-tolerant convergence functions. Though the overall number of concurrent processes and in particular the number of faulty processes is limited, our results are sufficient to argue dual fault tolerance as required by *TTEthernet*. A crucial aspect preventing better scalability is the number of SMT constraints generated which grows quadratically with the number of network components.

For the verification of complex problems, SAL provides guidance in the development of the proof by producing counterexamples. This is a practical and powerful feature that allows systematically strengthening of the invariant.

Although the formal verification of the full-blown *TTEthernet* clock synchronization service as a whole is outside the scope of this paper, the compression function as a core element will be used as a basic building block in future studies.

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 in part by NSF grant CSR-0917398 and by NASA Cooperative Agreement NNX08AC59A.

References

1. Steiner, W.: TTEthernet Specification, TTA Group (2008), `http://www.ttagroup.org`
2. Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K.: The time-triggered ethernet (tte) design. In: 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Seattle, Washington (May 2005)
3. Kopetz, H., Ochsenreiter, W.: Clock synchronization in distributed real-time systems. IEEE Trans. Comput. 36(8), 933–940 (1987)
4. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control 62, 190–204 (1984)
5. Pfeifer, H.: Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture. Ph.D. dissertation, Universität Ulm, Germany (2003), `http://www.informatik.uni-ulm.de/ki/Papers/pfeifer03-diss.pdf`
6. Pike, L.: Formal verification of time-triggered systems. Ph.D. dissertation. Indiana University (2006)
7. 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)
8. Brown, G.M., Pike, L.: Easy parameterized verification of biphase mark and 8N1 protocols. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 58–72. Springer, Heidelberg (2006)
9. Steiner, W.: TTEthernet Executable Formal Specification, Marie Curie Technical Deliverable RO_A (2009), Available via TTA Group, `http://www.ttagroup.org/`