Presented at the 30th IEEE/AIAA Digital Avionics Systems Conference (DASC), Seattle, WA, October 2011.
Recipient of "Best in Session" and "Best in Track" awards.

TTA and PALS: Formally Verified Design Patterns for Distributed Cyber-Physical Systems

Wilfried Steiner, TTTech Computertechnik AG, Vienna Austria
John Rushby, SRI International, Menlo Park CA

Abstract

Avionics systems in modern and next-generation airborne vehicles combine and integrate various real-time applications to efficiently share the physical resources on board. Many of these real-time applications also need to fulfill fault-tolerance requirements—i.e., the applications have to provide a sufficient level of service even in presence of failures—and this combination of real-time and fault-tolerance requirements elevates avionics to a class of cyber-physical systems of the highest complexity. Consequently, avionics design is challenging for avionics architects and application engineers alike.

One way to manage complexity is a division of the overall problem into a hierarchical set of layers connected by well-defined interfaces. The avionics architect may then select the fundamental network architecture, like AFDX, TTP or TTEthernet, and hide their idiosyncrasies—in particular, those concerning the way in which time is managed and presented—by providing a more uniform conceptual interface to the application engineer. In this paper we call this interface the "model of computation" and discuss the well-known synchronous model of computation and small extensions thereof for real-time systems. The bridge between the network architecture and the model of computation concerns the way in which distributed real-time applications are organized and it is achieved through "design patterns." We revise and formally analyze two such design patterns, the Sparse Timebase of the Time-Triggered Architecture (TTA) and the Physically-Asynchronous Logically-Synchronous (PALS) approach.

Perhaps surprisingly, we show that both design patterns rely on the same assumptions about the network architecture; hence, the choice of network architecture and design pattern should depend on pragmatics and formal considerations orthogonal to those required to support a particular model of computation. Our formal analysis builds directly on the verification in PVS of a PALS-like pattern for TTA that was developed 15 years ago, thereby illustrating that a mechanized formal analysis is an intellectual investment that supports cost-effective reuse.

1. Introduction

The design of distributed, fault-tolerant, real-time systems is a challenging topic—and an important one, too, as modern airplanes, automobiles, and many other kinds of domestic and industrial plants depend on such systems for their reliable and safe operation. Over time, a fairly systematic body of scientific knowledge has been developed on the topic, together with a body of engineering practice that builds, to a lesser or greater extent, on that knowledge. Central to the engineering practice is a layering of services that often corresponds to separate classes of products and suppliers. Typically, there is a network with its transmission media (e.g., wires or optical fibers), access controllers, and network hubs and switches, augmented by mechanisms to mask certain kinds of faults and sometimes to provide additional services such as clock synchronization. We refer to this as the *network architecture*; some network architectures are specified fairly abstractly, for example, the Time-Triggered Architecture (TTA) [8], whose current implementation is TTEthernet [19], while others are specified more concretely, for example, AFDX [1]. Application software that controls the physical plant uses the services of the network architecture but is generally designed against a more abstract interface that we refer to as a *Model of Computation* (MoC); an example of a MoC is the (round based) *synchronous system* model [9]. Designing to a MoC shields application software from the idiosyncrasies of a particular network architecture, but there must obviously be some mechanism to bridge the gap between these two interfaces. An MoC is more than a programming interface—it is a conceptual framework for organizing distributed real-time applications so that they operate as a coherent whole—and this organization is realized through "design patterns," which are the focus of this paper.

With the rising complexity of modern more-electric aircraft, more and more research and design groups realize that synchronized time is a key element of such design patterns, as it provides a way to coordinate the actions of spatially distributed nodes in a network. The TTA, for example, uses synchronous time in a design pattern called

the *sparse timebase* [5] in which nodes will generate critical events only during activity intervals that alternate with intervals of silence. In a proper instantiation of the sparse timebase, agreement protocols on time of occurrence of events can be avoided or at least greatly simplified. The PALS approach [18] is a more recent design pattern that builds on synchronized time. It is primarily focused on the coordinated exchange of messages in a network and utilizes synchronized time to specify intervals when messages may be sent and when not.

The PALS approach is similar to the sparse timebase, restricted so that transmission of messages is the only kind of event considered. However, PALS parameterizes its assumptions in a way that potentially makes this design pattern more suitable to native AFDX network architectures.

In this paper we will formalize and verify the design patterns that link TTA to refinements of the synchronous system MoC, and we will likewise formalize and verify the Physically Asynchronous Logically Synchronous (PALS) design pattern against the same MoCs. Along the way, we will correct some small errors in previous formalizations of properties of these patterns.

Our formalizations and verifications are undertaken in PVS [12] and build on a previous treatment that verified a design pattern linking the TTA network architecture to the synchronous system MoC [16] in a manner very similar to PALS. Thanks to the theory structuring capabilities of PVS, and the stability of its implementation, the formal development reported here does literally import and extend the formalization and proofs from 15 years ago (that earlier work was performed in 1996 and first reported in 1997). This corroborates previous evidence [15] that a mechanically supported formal verification is an intellectual investment that can yield cost-effective benefits beyond its original purpose.

The structure of the paper is as follows. In Section 2, we recapitulate the previous formalization of the synchronous system MoC and extend it a little with explicit timing information; we also recall properties of clock synchronization and minimal synchrony assumptions. In Section 3, we describe how timestamps and a π/Δ-precedent event set allow a distributed system to establish the temporal order of events, and the application of this idea in TTA as a sparse timebase. We describe how this approach can be used to represent the synchronous system MoC in TTA. We then present a formal assessment of this approach and formally verify (corrected versions of) the four "fundamental limits of time measurement" that underpin the TTA sparse timebase. In Section 4, we observe that the two phases of a round in the synchronous system MoC (messaging, followed by computation) vitiate some of the benefits of the sparse timebase, since it is necessary to wait for messages to be delivered before proceeding to the computation phase. In this case, messages are spaced sufficiently that timestamps are no longer necessary, provided there is a delay at the start of each round to ensure that all nodes are in the same round. This was the design pattern verified by Rushby [16]. We then introduce PALS, which is the same approach, but using a different parameterization of the network architecture. We also present a formal assessment of the PALS design pattern and formally verify (corrected versions of) its properties. We conclude in Section 5 and discuss further work.

2. Models of Computation

In our terminology, a distributed system is composed of nodes and communication channels. Nodes have inputs, outputs, and state, and can change their outputs and state as a function of their inputs and their current state. Nodes are connected to each other via communication channels (of unspecified topology) that are used to exchange information in the form of messages.

A model of computation (MoC) describes how the state transitions of individual nodes are coordinated and, hence, how the global state of the system evolves. The *synchronous system* MoC is one in which the system evolves in a series of "rounds" [9]. Rounds have two phases: in the first, each node in the system sends a message to some or all of the other nodes (different messages may be sent to different nodes; messages depend on the current state of the sender); in the second phase, each node changes its state in a manner that depends on its current state and the collection of messages it received in the first phase. There is no notion of real-time in this model: messages are transferred "instantaneously" from senders to recipients between the two phases. The nodes operate in lockstep (i.e., they are *synchronous*): all of them perform the two phases of the current round, then move on to the first phase of the next round, and so on.

The synchronous system MoC allows a distributed system to be viewed as a coordinated whole and therefore aids the design of distributed algorithms. This MoC is sufficiently expressive to support the description of classical computer science algorithms such as consensus, agreement, and membership [9] and it is an attractive framework in which to develop applications for distributed systems. Rushby [16] illustrates how programs for the synchronous system MoC can be mechanically verified by interpreting them as functional programs, and Meseguer and Ölveczky [10] likewise use such programs as the starting point for their analysis.

2.1. Synchronous System MoC

Some elements from our PVS formalization of the synchronous system MoC, taken directly from Rushby's earlier treatment [16] as corrected by Pike [14], are summarized below. We do not describe the formalization in detail here, since it is available in the cited references; we simply introduce the notation and concepts that we will need later. The PVS "dump" file that contains the commented PVS sources and proofs for this paper can be found online[1]. The guidelines on how to extract the PVS sources as well as to re-execute the proofs can be found in the PVS user guide [13].

```
proc, mess, state: TYPE+
p, q: VAR proc
in_nbrs, out_nbrs: [proc -> setof[proc]]
```

PVS is a dependently-typed higher-order logic with predicate subtyping [17]. Here, `proc` specifies the nodes (also called the "processors") in the system as a nonempty type; `mess` and `state` similarly specify the messages and the local state of a node. The higher-order functions `in_nbrs` and `out_nbrs` define the input neighbors and output neighbors of a given node.

In the synchronous system MoC all nodes change their state and send and receive messages in lock-step. An execution of a distributed system in the synchronous MoC, as introduced in [9, p. 20], is thus represented as follows:

$$(S_0, M_0, N_0), (S_1, M_1, N_1), (S_2, M_2, N_2), \ldots \quad (1)$$

where S_i represents the global system state at the start of round i, M_i are the messages sent in round i, and N_i are those received in round i (M_i and N_i may differ in the presence of faults).

Although the synchronous system MoC is adequate for the description of many distributed algorithms, it is not expressive enough to support descriptions of cyber-physical applications, such as the control laws for a physical plant, because it provides no access to real time. The following two MoCs are simple extensions to the synchronous system MoC that provide capabilities to observe and measure real time.

In addition to their intrinsic interest and utility, these extensions also serve to introduce the formalizations of clocks that we will need in our later verifications.

2.2. Time-Aware MoC

The *time-aware* MoC is a minimal extension to the synchronous system MoC that adds a real-time clock to the state of each node. The clock provides a counter that ticks at a rate proportional to real time. Thus, algorithms "running on" the time-aware MoC can measure the passage of real-time.

A clock can be formalized independently of its physical realization as a function that relates real time to the value of a counter.

```
realtime: TYPE = nonneg_real
clocktime: TYPE = nat
t: VAR realtime
C(p, t): clocktime
```

Here, `realtime` represents a global notion of time available to an omniscient observer; it is a continuous quantity, formalized as a nonnegative real number. On the other hand, `clocktime` represents the simulation of time within each node: it is the value of a counter. The clock of node p is a function `C(p, t)` that returns the value of p's clock counter at `realtime` t.

The quality of a real-time clock is a measure of how accurately `C(p, t)` tracks real time. The quality parameter we use is called the *drift rate*, denoted by ρ, which takes values between 0 and 1 (it will be very close to 0 for a good clock).

```
rho: {x: real | 0 < x AND x < 1}
drift_rate: AXIOM t1 >= t2 IMPLIES
 floor((1-rho)*(t1-t2)) <= C(p,t1)-C(p,t2) AND
 C(p,t1)-C(p,t2) <= ceiling((1+rho)*(t1-t2))
```

The axiom `drift_rate` ties `realtime` to `clocktime`: given any two points in `realtime`, $t_1 \geq t_2$, the difference in the clock counter values of an arbitrary node p at times t_1 and t_2 is between $(1 - \rho)$ and $(1 + \rho)$ times the realtime duration $t_1 - t_2$, rounded appropriately. Notice this requires that `clocktime` and `realtime` both use the same units to measure time (e.g., microseconds).

It is now easy to prove a lemma on the monotonicity of a clock: an increase in `realtime` implies a monotonous increase in `clocktime`. Notice that, depending on the length of the `realtime` interval, the clock may or may not advance.

```
monotone_clock:
   LEMMA t1 < t2 => C(p, t1) <= C(p, t2)
```

In the time-aware MoC, different nodes can accurately measure the passage of real time: for any given real-time duration, the clock counters in different nodes will advance by approximately the same amounts (it is approximate because they may have different drift rates). However, their actual clock counter values may be quite different (one may advance from 34 to 87, another goes from 109 to 162). Hence, we define an enhanced MoC in which these values are synchronized.

1 `http://www.csl.sri.com/users/steiner/time_synchronized_systems.dmp`

2.3. Time-Synchronized MoC

Here, the real-time clocks introduced in the time-aware MoC are synchronized to each other, so that at any point in `realtime` the `clocktime` in all nodes is about the same. The maximum difference is called the "precision" Π of the distributed system and we express the upper bound on Π by Σ with $\Pi < \Sigma$. The synchronized clocks are formalized by an additional axiom on `C(p, t)`.

```
Sigma: clocktime
clock_sync: AXIOM abs(C(p,t) - C(q,t)) < Sigma
```

It is important to understand that the synchronous system MoC places very strong requirements on its underlying network architecture. A landmark result [4] established that distributed consensus is impossible in an "asynchronous" system in the presence of even a single fault; later work examined the "minimal synchronism" needed for consensus [3]. Since the synchronous system MoC can perform consensus, its underlying network architecture must provide at least this minimal synchronism, which is the existence of a guaranteed upper bound on message delays between nonfaulty nodes.

This is formalized as follows. We use `sent(p,q,m,t)` to indicate that node p sent message m to node q at real time t and we use `recv(q,p,m,t)` to indicate that node q received message m from node p at real time t. These two events are related as follows (`out_nbrs(p)` is the set of outgoing channels of node `p`, the parenthesis `(out_nbrs(p))` lifts this to a predicate subtype).

```
min_latency: {x: realtime | 0 < x}
max_latency: {x: realtime | min_latency <= x}
latency: TYPE = {x: realtime | min_latency <=x
                          AND x <= max_latency}
max_delay:
 AXIOM FORALL p, (q: (out_nbrs(p))), m, t:
  (EXISTS (d: latency): sent(p, q, m, t)
                        => recv(q, p, m, t + d))
  AND (EXISTS (d: latency): recv(q, p, m, t)
                        => sent(p, q, m, t - d))
```

Off the shelf network technologies such as CAN Bus or Ethernet cannot provide the guarantee represented by `max_delay`. The reason is that faulty nodes can interfere with nonfaulty ones: a faulty node that does not follow the protocol by, for example, transmitting constantly or transmitting on top of the messages from some nonfaulty node, can lead to unbounded delays. The only way to control this possibility is to add redundancy and have a second component mediate each node's access to the network medium. This "network guardian" can limit the rate at which each node can transmit, or restrict the specific times at which it is allowed to transmit. The former scheme is used in AFDX while the latter, which requires a global schedule, is used in TTA. Use of TTA requires more upfront effort than AFDX (to establish the global schedule) but its maximum message delay is generally smaller.

We now examine how suitable design patterns enable specific network architectures to support the MoCs.

3. Time-Triggered Architecture

3.1. Informal Overview

The TTA uses a global schedule to control access to the network. Consequently, each node of the network architecture has a synchronized clock and it is straightforward to reflect this in the MoC; hence these aspects of the time-aware and time-synchronized MoCs are easily achieved. The underlying synchronous system MoC is more challenging. This is because the clocks are not exactly synchronized, so if we arrange to start a new round every hundred clock ticks, say, some nodes will start the round before others and there will be some confusion which round the global system is in. Figure 1 illustrates this problem: node A and node B are synchronized with precision Π. As it happens in this scenario, both nodes perceive an event $e1$ (e.g., a message broadcast by a third node) that occurs at about the start of a new round. However, as node A has a slightly faster clock than node B, node A assigns $e1$ to round $i+1$ while node B assigns $e1$ to round i.

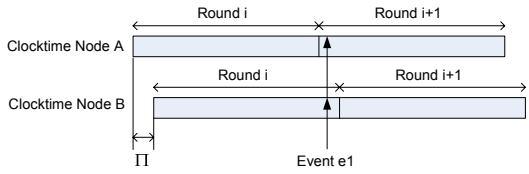

Figure 1: Imperfection of clock synchronization causes imperfect event-to-round assignment

There are two approaches to dealing with this problem: one uses timestamps and the notion of a sparse timebase, while the other adds delays to ensure that all nodes are in the same round.

The first approach builds on TTA design patterns for "global time" and a "sparse timebase," depicted in Figure 2. These design patterns are major elements in constructing the Time-Triggered MoC [7]; here, we use them to construct the synchronous system MoC.

An analogy (due to Kopetz) may help motivate the idea of a sparse timebase. Imagine we have a train track along which a train passes every hour, on the hour. By looking at a watch, we can correctly identify any train (e.g., "that's the 2 o'clock train"), and this remains true even

if trains do not keep perfectly to the timetable (e.g., they may be up to 5 minutes early or late) and even if our watch does not keep perfect time (e.g., it may be up to 5 minutes fast or slow). But we will not be able to correctly identify trains in the face of these imprecisions if they are scheduled more frequently: say, every 15 minutes. This method only works if trains are "sparse" relative to the timing errors present in the system.

In a distributed system, where network delays may be quite large compared to other time uncertainties in the system, we can help other nodes interpret reports of our observations (e.g., our train sightings) by sending them with our timestamp, rather than having receiving nodes record their local time when our report is received. For global consistency, we require that all nodes assign the same temporal order to all reported events, and one way to make this feasible is to require that events are sparse.

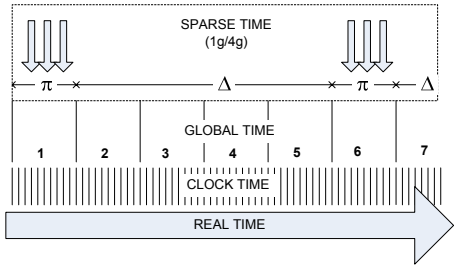

Figure 2: Overview of the global time and sparse timebase definition

These ideas are realized in the TTA by first defining "global time" as is a coarser time layer on top of `clocktime` (see Figure 2). This time layer groups a specified number of `clocktime` ticks into a single tick of global time. A global time is defined to be *reasonable* [6] when the duration of one tick in the global time is longer than the precision of the system (and we generally assume that the precision is longer than a tick in `clocktime`). This constraint on tick duration only as a function of the precision in the system contrasts with PALS (see next section), where it also depends on other system's properties (such as transmission latencies, and computation or queueing overheads).

In the following, we assume a reasonable global time of *granularity* (i.e., tick duration) g. Without loss of generality, we set the $g = \Sigma$ (since Σ is an arbitrary value with $\Pi < \Sigma$). Events, which can occur at any instant on a dense timeline (i.e., in `realtime`), may be recognized by nodes and a node will assign each event (that it recognizes) a timestamp according its local view of the global time (i.e., the node's current global time value). A node may then send the information that it recognized an event together with its assigned timestamp to other nodes. In such a setup, the following informally stated "Fundamental Limits of Time Measurement (FLTM)" [6, p. 55] apply:

> **FLTM (i)**: If a single event is observed by two different nodes, there is always the possibility that the timestamps differ by one tick. Hence, a one-tick difference in the timestamps of two events is not sufficient to reestablish the temporal order of the events from their timestamps.
>
> **FLTM (ii)**: If the observed duration of an interval is d_{obs}, then the true duration d_{true} is bounded by $(d_{obs} - 2g) < d_{true} < (d_{obs} + 2g)$
>
> **FLTM (iii)**: The temporal order of events can be recovered from their timestamps, if the difference between their timestamps is equal to or greater than 2 ticks.

Now, the philosophy in TTA is that events are triggered by a schedule, so the question becomes, given that we can influence the generation times of events, when shall they be generated such that temporal ordering is always possible. In analogy to Verissimo's δ-precedence scheme [21, p. 43 ff], Kopetz defines π/Δ-precedence[2]: a set of events e_i is said to be π/Δ-precedent, if, and only if, any two events e_1, e_2 of the event set are generated either within π time units or are at least Δ time units apart. Figure 2 shows an example of an π/Δ-precedent event set. From the π/Δ-precedence scheme, the fourth FLTM [6, p. 55] follows:

> **FLTM (iv)**: The temporal order of events can always be recovered from their timestamps, if the event set is $0/3g$-precedent.

Kopetz [5] builds the concept of a sparse timebase on the π/Δ-precedence scheme, but chooses the parameters $1g/4g$. The reason for this adjustment is that π and Δ are `realtime` parameters, but all events in a time-triggered system are generated by the nodes themselves, according to a schedule that operates on global time, which is derived from local `clocktime`. Because their local clocks are not perfectly synchronized, nodes cannot guarantee to generate events that are truly simultaneous (i.e., $\pi = 0$); the best that can be done is to generate events at the same global time, but these may then be as much as Σ apart in `realtime`. Hence, the parameter π is set to $1g$ (which is no less then Σ) and Δ is correspondingly adjusted to $4g$.

2 Note, that this π is different to the Π representing the precision.

Sparse Timebase: The temporal order of events can always be recovered from their timestamps, if the event set is at least $1g/4g$-precedent.

A $1g/4g$-precedent sparse timebase is sufficient to establish the synchronous system MoC: nodes transmit their messages within the same global clock tick and spend at least $4g$ global clock ticks in the computation phase. The reception of messages is performed continuously by each node's operating system and each message received is placed in the input buffer of the appropriate round by examination of its timestamp (even if it arrives before the receiving node has started that round or, due to network delays, after it has finished its own messaging phase). Although messages can be correctly allocated to rounds with a $1g/4g$ timing of the messaging/computation phases, messaging delays may mean that this can only be done retrospectively (e.g., if the message delay is greater than $5g$); since the synchronous system MoC requires that all incoming messages are available before the computation phase begins, the start of this phase must be postponed for at least the maximum message delay. Hence the duration of a round must be at least $5g$ units, but will generally be longer than this to allow for message delays and the time required to actually perform the computation phase.

Correctness of the sparse timebase construction, and of the design pattern that uses it to support the synchronous system MoC rests on the four FLTM claims. We formalize and verify these in the next subsection.

3.2. Formal Assessment

Our formal treatment defines `GLOBALtime` as a timing layer on top of `clocktime`. This timing layer groups a certain number of `clocktime` ticks into a single tick of `GLOBALtime`. The TTA defines this grouping operation as a function only of the precision in the system, rather than also depending on the system's latencies (transmission, computation, queueing). Specifically, the `granularity` of the `GLOBALtime` has to be at least `Sigma`, i.e., longer than the worst-case offset of any two clocks (this is the "reasonableness condition").

```
GLOBALtime: TYPE = nat
granularity: clocktime = Sigma
G(p, c): GLOBALtime = floor(c/granularity)
```

Given the formal definition of the `GLOBALtime` layer, we can formulate FLTM (i) as theorem on the difference of the `GLOBALtime` timestamps as taken by any two different clocks p and q of an event that happened at time t. According to FLTM (i) there may be a difference of up to one `GLOBALtime` tick. Figure 1 depicts FLTM (i) in case that the round length is equal to one `GLOBALtime` tick. We have verified FLTM (i) in PVS by the following theorem.

```
FLTM_i: THEOREM (FORALL (t: realtime):
    G(q, C(q, t)) - G(p, C(p, t)) = 0 OR
    G(q, C(q, t)) - G(p, C(p, t)) = 1 OR
    G(q, C(q, t)) - G(p, C(p, t)) = -1)
```

FLTM (ii) discusses the quality of time measurement for an interval $[t_a, t_b]$. Our formal treatment shows a slight imprecision in FLTM (ii): the time interval is not only bounded by $+/-$ two times the granularity of the `GLOBALtime`, but also a function of the drift rate ρ of the real-time clocks. Figure 3 illustrates this dependency; it plots `clocktime` against `realtime` and depicts two nodes A and B with fast clocks. Here, a perfect clock is illustrated by a line with forty-five degree slope and in contrast, the fast clocks are depicted by lines with slopes of more than forty-five degrees. The figure suggests that the length of d_{obs} is a function of the slopes for nodes A and B and we can formally verify this hypothesis with PVS.

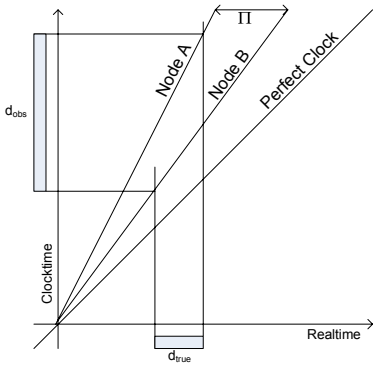

Figure 3: Example of the influence of drift rate rho on the measurement of realtime intervals

In our formal framework this dependency is reflected in the time-synchronized MoC, where the upper bound that defines clock synchronization is defined as a constraint in `clocktime` and not `realtime`. The relation between these two times is defined in the time-aware MoC by the parameter ρ. Time measurement has, therefore, to account for a factor `(1+rho)` and `(1-rho)` in the bounds on $[t_a, t_b]$.

Note that we have retained a limitation of the original FLTM (ii), which is that clocks are not synchronized to an external time reference such as GPS. Hence, all the clocks in the system may collectively drift from `realtime`. It is straightforward to adjust both the formalization and any implementation so that synchronization is performed to an

external time reference, and thereby bounds the common drift. This is done in PALS: see Section 4.1.

```
FLTM_ii: THEOREM (FORALL (t_a, t_b: realtime):
t_b >= t_a =>
    ((G(q,C(q,t_b))-G(p,C(p,t_a)))*granularity
      - 2*granularity
      <= ceiling((t_b-t_a)*(1+rho)))
AND (floor((t_b-t_a)*(1-rho)) <=
     (G(q,C(q,t_b))-G(p,C(p,t_a)))*granularity
      + 2*granularity))
```

FLTM (iii) says that a difference of two in the timestamps of two events according the `GLOBALtime` is sufficient to order the events. This directly follows from FLTM (i) and we have verified FLTM (iii) by following theorem.

```
FLTM_iii: THEOREM (FORALL (t_a,t_b: realtime):
  (G(q,C(q,t_b))-G(p,C(p,t_a))>=2) => t_a<t_b)
```

Two events that happened at `t_a` and `t_b` are perceived by two nodes `p` and `q`, which assign these events timestamps according the global time `G`. When the difference in these timestamps is at least two, then the event at `t_a` happened before `t_b`.

For FLTM (iv) and the sparse timebase, we define the event set by a nonempty type and specify two particular events `e1` and `e2`.

```
events: TYPE+
e1, e2: VAR events
```

$0/3g$-precedence requires that any two events occur either at the same point in `realtime` or at least $3g$ (three times the granularity of the global time) apart. In analogy to FLTM (ii), our formal investigation shows that this definition is imprecise: the $3g$ requirement assumes perfect clocks. Hence, $3g$ in `realtime` may take up to $\frac{3g}{(1-\rho)}$ in `clocktime`.

We formulate FLTM (iv) using `z()` as an uninterpreted function that assigns to each event its time of occurrence in `realtime`.

```
z(e1): realtime
FLTM_iv: THEOREM
 z(e1) < z(e2) AND
 floor((z(e2)-z(e1))*(1-rho)) >= 3*granularity
 => G(p, C(p, z(e2))) - G(q, C(q, z(e1))) >= 2
```

Two events that are $0/3g$-precedent and occur at different points in `realtime` will always cause their timestamps to differ by at least two ticks. (By FLTM (i), those that occur at the same point in `realtime` will have timestamps that differ by at most one tick.)

As explained in the informal overview, the sparse timebase adjusts this treatment to use $1g/4g$-precedence. Here, two events may either occur within $1g$ or at least $4g$ apart. Again, we have to normalize both intervals from `realtime` to `clocktime`: the $1g$ upper bound becomes $\frac{1g}{(1+\rho)}$ while the $4g$ lower bound becomes $\frac{4g}{(1-\rho)}$. Again, we use the uninterpreted function `z()` to represent when events happened according to `realtime`.

```
sparsetime_a: THEOREM
z(e2) >= z(e1) AND
ceiling((z(e2)-z(e1))*(1+rho)) <= granularity
=> G(p, C(p, z(e2))) - G(q, C(q, z(e1))) <= 2
```

```
sparsetime_b: THEOREM
z(e2) >= z(e1) AND
floor((z(e2)-z(e1))*(1-rho)) >= 4*granularity
=> G(p, C(p, z(e2))) - G(q, C(q, z(e1))) >= 3
```

The corrections on FLTM (iv) for the sparse timebase require the events to happen closer together or slightly farther apart than a multiple of g. Hence, if we want to express π/Δ-precedence in multiples of g we would have to increase this from $1g/4g$ to $1g/5g$-precedence, which is quite inconvenient. A correction that leaves FLTM (iv) and the sparse timebase in their original form would therefore be beneficial, and, indeed, we can compensate for the imprecisions by moving the corrections into a re-definition of the reasonableness condition.

```
granularity_c(number_granules): clocktime
reasonableness_condition: AXIOM
 rho < 1/number_granules AND
 granularity_c(number_granules) =
    ceiling((Sigma+1)/(1-number_granules*rho))
Gc(number_granules, p, c): GLOBALtime =
    floor(c/granularity_c(number_granules))
```

While originally the granularity has been defined as a function of Σ only, the new version defines granularity as a function of Σ, ρ, and the `number_granules` (given as $\max(\pi, \Delta)$ of the required π/Δ-precedence). Given the granularity $g = \lceil \frac{\Sigma+1}{1-number\_granules \times \rho} \rceil$, we can formulate a generalized form of FLTM (iv):

```
FLTM_iv_general: THEOREM
 z(e1) < z(e2) AND rho < (1/number_granules)
 AND number_granules >= 2
 AND z(e2)-z(e1) >=
 number_granules*granularity_c(number_granules)
 => Gc(number_granules,p,C(p, z(e2))) -
    Gc(number_granules,q,C(q, z(e1)))
     >= (number_granules-1)
```

The above theorem can be instantiated for FLTM (iv) and for `sparsetime_b`, by setting `number_granules` to 3 and 4 respectively. For `sparsetime_a`, we get the following alternative constraint when using the re-defined granularity.

```
sparsetime_a_alt: LEMMA
 z(e1) < z(e2) AND rho < (1/4)
 AND z(e2)-z(e1) <= granularity_c(4) =>
 Gc(4,p,C(p,z(e2))) - Gc(4,q,C(q,z(e1))) <= 2
```

4. Physically Asynchronous Logically Synchronous

4.1. Informal Overview

We have seen that a $1g/4g$-precedent sparse timebase allows construction of a general timestamping service whose accuracy depends only on system precision (clock skew) and not on other parameters such as message delays. However, the two-phase nature of the synchronous system MoC requires nodes to wait for the maximum message delay before proceeding from the messaging to the computational phase of each round. Given this wait, we can dispense with timestamps, provided we also wait for a short period at the start of each round before beginning the messaging phase (this is to ensure that every node has started the round). Rushby [16] describes and formally verifies this pattern. He shows that the delay D at the start of each round must be at least Σ (Rushby assumes the minimum latency is zero) and that the computation phase can start no earlier than $D + \Sigma + (1 + \rho)\delta$, where δ is the maximum latency.

PALS [18] is a similar pattern developed 10 years later; it differs from Rushby's TTA scheme chiefly in its choice of network parameters: for example, Rushby has only a maximum network latency, whereas PALS has both minimum and maximum latencies. (A minimum latency of zero is appropriate for TTP bus networks, which was the network architecture assumed by Rushby, whereas nontrivial minimum latencies are appropriate for AFDX networks, as considered for PALS.)

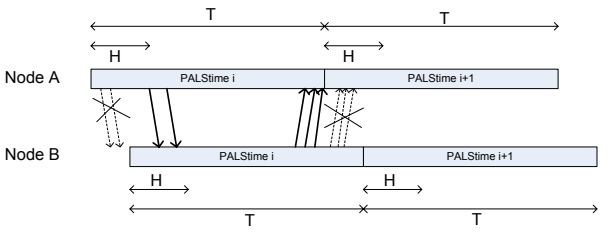

Figure 4: PALS overview

An overview of PALS is given in Figure 4. PALS defines a time layer similar to the `GLOBALtime` of TTA that we formalize as `PALStime` and refer to as PALS time or the PALS clock. Figure 4 depicts two nodes A and B for a duration of two ticks in `PALStime`. Node A is slightly faster than node B and, thus, the PALS clock of node A will tick a little bit earlier than node B's PALS clock. However, the PALS clocks of A and B are synchronized, such that it is guaranteed that both clocks will tick within a known interval. Furthermore, Figure 4 also depicts arrows between the nodes A and B, which are intended to illustrate message exchange between the nodes and the direction of the arrows represents the direction of transmission. Some of the arrows are crossed out, meaning that the respective transmissions are not allowed at the depicted points in time and PALS specifies these intervals of unallowed transmissions using the parameters H and the T. The H parameter specifies a timeout of silence immediately after the tick of the PALS clock. It ensures that a node with a fast PALS clock will send messages only at times when it is guaranteed that receiving nodes, which may have slower PALS clocks, have the same `PALStime`. Likewise, the T parameter ensures that the duration of a PALS clock tick is sufficiently long that messages sent by nodes with slow PALS clocks are received by nodes with fast PALS clocks within the same `PALStime`. In the following we review the PALS design pattern as presented in the literature and then formally verify that PALS provides the synchronous MoC.

PALS requires minimal synchrony assumptions, which are formulated as conditions "G2" (bounded transmission delay), "G3" (bounded processing time), and the definition of PALS Clocks (bounded clock drift and synchronized clocks) [18]:

> **G2: Real Time Network.** The network has a network queueing (scheduling) delay q bound by $0 < q_{min} \leq q \leq q_{max}$ and a network transmission delay μ bounded by $0 < \mu_{min} \leq \mu \leq \mu_{max}$.
>
> **G3: Real Time Machine.** ...The task completion time α, including real time scheduling, computation, and I/O is bounded by $0 < \alpha_{min} \leq \alpha \leq \alpha_{max}$ ...
>
> **PALS Clocks.** All the local clocks used by PALS for global computation are synchronized with the global clock with skews of at most ϵ.

Note that the global clock of PALS is a different concept than the global time of the sparse timebase: the former is a concrete external time reference (e.g., GPS), while the latter is an abstraction constructed separately in each node. The clock synchronization of the sparse timebase is *internal* synchronization: the local clocks of all nodes remain close together, but they may collectively drift from any external reference. The clock synchronization of PALS is *external* synchronization: the local clocks of all nodes are kept close to the external reference. Clearly, external synchronization implies internal and, in fact, is often used to implement it.

We denote the ϵ of "PALS Clocks" by ϵ_{PALS}. In the terminology of Kopetz ϵ_{PALS} is the "accuracy" of the

system, which also defines the precision Π of a system: $\Pi = 2 \times \epsilon_{PALS}$. Precision, as before, is the maximum offset of any two non-faulty local clocks in the system.

It is the aim of PALS that all nodes (called "real-time machines" in PALS) process their input synchronously. In PALS terminology this is expressed as: all messages sent at time j, according a sender's PALS clock C_s, will be received also at time j, according each receiver's PALS clock C_r. At PALS time $j+1$ each real-time machine will then process the inputs received during PALS time j. This synchronous input processing is established by restricting the intervals when a real-time machine is allowed to send messages.

Although not explicitly stated in the PALS model, we claim it is justified to assume that the global time used as reference time ticks with a granularity of T and the tick is synchronized with the PALS clocks with an accuracy of ϵ_{PALS}. Hence, whenever the global time ticks, a PALS clock may tick up to ϵ_{PALS} later or has ticked less than ϵ_{PALS} earlier.

The PALS design pattern restricts the points in time when messages are allowed to be sent by condition "G6":

> **G6: PALS Causality Rule.** A machine at (PALS) clock period j cannot send earlier than $\uparrow (C_i = j) + H$, where $H = 2\epsilon_{PALS} - \mu_{min}$.

$\uparrow (C_i = j)$ indicates the event when PALS clock C_i ticks to time j. The PALS Causality Rule ensures fast PALS clocks will not cause messages to be received at $j-1$ by machines with slow PALS clocks. In order to also ensure that messages sent by machines with slow clocks at j will not be received by machines with fast clocks at $j+1$, the granularity of the global time, T, has to be sufficiently long:

> **G7: PALS Clock Period.** PALS clock period $T > 2\epsilon_{PALS} + \max(\alpha_{max} + q_{max}, H) + \mu_{max}$

The system assumptions defined in G2, G3, and the definition of PALS clocks, together with the PALS design pattern defined in G6 and G7 should ensure that all real-time machines process the system-internal messages at the same PALS clock tick. Formally, PALS is intended to satisfy the following theorem:

> **Fact 3.** A message sent during sender's j^{th} clock period will be received by all machines when they are still in their j^{th} clock period.

We have formalized the proof of Fact 3, based on the PALS assumptions, in PVS. As we will discuss in the following subsection, our formal proofs verify the general concept, but also identified some imprecisions in PALS.

4.2. Formal Assessment

We build the formal assessment of PALS on the time-synchronized model (Section 2.1). We start with the definition of an additional time layer `PALStime` on top of `clocktime`.

```
PALStime: TYPE = nat
PALS_period: clocktime
P(p, c): PALStime = floor(c/PALS_period)
```

`PALStime` is represented by the natural numbers. The PALS clock ticks with a period `PALSperiod` measured in `clocktime`. The function `P(p,c)` returns for each processor p and each `clocktime` c the current `PALStime`. As depicted, the `PALStime` is simply the integer division of the c by the configured `PALSperiod`. We continue the discussion with the formalization of G6 and G7.

```
sent_PALS: AXIOM
(FORALL p, (q: (out_nbrs(p))), m, t:
 sent(p, q, m, t) =>
  rem(PALS_period)(C(p, t)) >
   Sigma - floor(min_latency * (1 - rho)) AND
  rem(PALS_period)(C(p, t)) <
   max(Sigma - floor(min_latency * (1 - rho)),
       max_task_time + max_queue_delay))
```

G6 prevents messages from being sent at the beginning of a PALS round. We can formally express this using the remainder function, which is formalized in the PVS library as the higher-order function `rem`: messages may only be sent when the remainder of the integer division of a node p's current `clocktime` by the `PALSperiod` is sufficiently high. Note, that Σ bounds the maximum difference of any two nodes in the system, hence $\Sigma > 2 \times \epsilon_{PALS}$.

In formalizing PALS, we identified two issues in the original presentation of PALS [18]. First, PALS does not explicitly distinguish between `realtime` and `clocktime`. This leaves the PALS definition slightly inaccurate, leading to potential scenarios as for example the one portrayed in Figure 5.

The depicted scenario shows, again, two nodes A and B where A has a faster PALS clock than B. On the bottom of Figure 5 the timing parameters as presented in [18] are illustrated: the PALS clocks of A and B tick with a distance of $2 \times \epsilon$ in `realtime`; then, when perfect clocks are assumed, waiting for the H parameter ensures that the transmissions of node A will arrive at node B during the same PALS clock tick. However, on top of Figure 5 we consider non-perfect clocks. In case that the local timer that measures the H timeout is slightly faster (indicated by $-\rho$) than the perfect clock, the timeout will expire too early and messages may be sent too early and received by node B while its PALS clock is still at `PALStime` $i-1$. The PALS formulation overlooks the fact that we must ac-

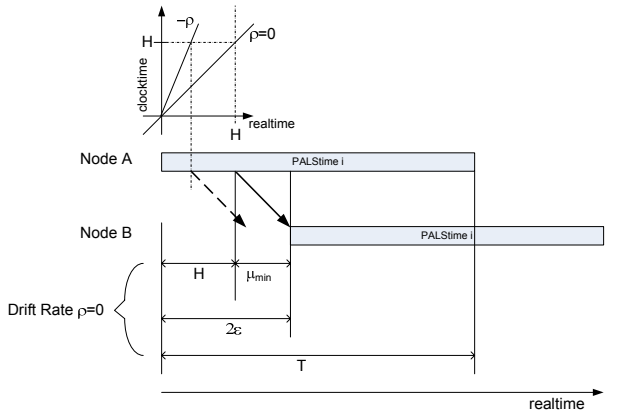

Figure 5: Improper treatment of realtime as clocktime potentially leads to PALS protocol violation

count for clock drift whenever we use local clocks to measure a real time interval.

Specifically, we see that μ_{min} (`min_latency`) and ϵ are durations in `realtime` and have to be normalized to `clocktime`. As discussed earlier $2\times\epsilon$ can be expressed by Σ in our framework and μ_{min} can be transformed by considering the drift rate with the factor `(1-rho)` (see Section 2.2 for discussion of the drift rate ρ).

Although one may argue that per definition the PALS clocks require that all "local clocks for global computations" are synchronized to the perfect clock with no more than ϵ deviation, still Figures 2-4 in [18] indicate that the PALS clocks of two nodes may tick up to $2 \times \epsilon$ time units apart. Without casting μ_{min} from `realtime` to `clocktime`, the PALS clock of the sending node would not be allowed to drift during the measurement of the H duration. Our formalization describes, how μ_{min} can be transformed such that scenarios like the one discussed above are mitigated.

G7 restricts the minimum length of `PALSperiod`. Here the normalization factor `(1+rho)` has an even higher impact, as the maximum latency has to be normalized.

```
PALS_period_min: AXIOM
 PALS_period >
 Sigma + max(Sigma-floor(min_latency*(1-rho)),
             max_task_time + max_queue_delay)
 + ceiling(max_latency*(1+rho))
```

The second finding has been corrected in later publications of the PALS design pattern [10] and [11], but as these publications reference the original one without calling out the shortcoming explicitly, we believe that the following discussion is still appropriate. There is a restriction on the precision Σ in the system: $H = 2\epsilon_{PALS} - \mu_{min}$ has to be greater than or equal to 0, as a negative H breaks the PALS pattern (sender would have to send at `PALStime` $j-1$ to cause a reception at `PALStime` j). We formalize this restriction as an additional axiom.

```
latency_aux6: AXIOM
  Sigma > floor(min_latency * (1 - rho))
```

This restriction means that in order to execute PALS, a system must not synchronize its clocks better than its minimum transmission latency. This is quite a significant handicap as the minimum latency easily reaches tens or hundreds of microseconds (considering a 100 Mbit/s store-and-forward Ethernet network). Luckily, this restriction can easily be removed by having $H = max(0, 2 \times \epsilon_{PALS} - \mu_{min})$.

The formalization developed for PALS allows us to prove Fact 3 of PALS: transmission and reception of a message will always happen at the same `PALStime` j. The verification of Fact 3 is eased by the general reasoning on the equality of integer divisions of two dividends with respect to the same divisor as formulated in the following lemma.

```
floor_equ: LEMMA
(FORALL (a, b: integer), (x, y: real),
         (base: posnat):
 a >= b - x AND a <= b + y
 AND rem(base)(b) - x > 0
 AND rem(base)(b) + y < base
     => floor(a/base) = floor(b/base))
```

`floor_equ` defines preconditions to be met so that the floor of the integer divisions of two dividends with respect to a common divisor are equal. We then show that PALS meets these preconditions which is the core of the formal proof of Fact 3.

```
recv_PALS: THEOREM
 (FORALL p, (q: (out_nbrs(p))), m, t:
  (EXISTS (d: latency): sent(p, q, m, t)
   => recv(q, p, m, t+d) AND
      P(p, C(p, t)) = P(q, C(q, t+d))))
```

Fact 3 is the key theorem to prove the equivalence of PALS to the synchronous system MoC. The proof can be found in our online PVS sources and follows Rushby's approach for time-triggered systems [16].

5. Conclusion

We have formalized and verified two design patterns that allow suitable network architectures (i.e., those satisfying minimal synchrony assumptions) to support more abstract models of computation, based on the synchronous system MoC, that shield applications programs from the underlying complexities.

Both TTA and PALS define time layers on top of synchronized real-time clocks: `GLOBALtime` and `PALStime`,

respectively. However, they differ with respect to purpose and generality. TTA `GLOBALtime` is a generic service to timestamp and to trigger events within a sparse timebase. The length of a tick in `GLOBALtime` is driven only by the precision of the system and the quality of its real-time clocks, not by message delays or other parameters. However, the two-phase character of the synchronous system MoC forces a wait (for messages to arrive) between its messaging and computational phases and this vitiates some of the benefits of the sparse timebase. Thus, although the sparse timebase can support the synchronous system MoC, it is perhaps best seen as a distinct MoC in its own right.

On the other hand, a clock tick in `PALStime` has a one-to-one relation to the notion of a round in the synchronous MoC. The PALS design pattern specifies the tick duration such that it is sufficiently long for transmission, reception, and processing a round of messages, accounting for all clock skews and message delays. Figure 6 depicts this relation between `PALStime` (on the top) and `GLOBALtime` (on the bottom). Here, one tick in `PALStime` correspond to twenty ticks in `GLOBALtime`.

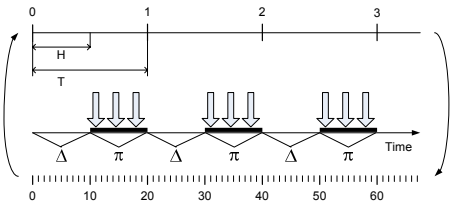

Figure 6: Comparison of `PALStime` and `GLOBALtime` and the PALS relation to π/Δ-precedence

Our formalization and verification revealed issues in previous presentations of the TTA sparse timebase and PALS. In both cases, certain expressions relating clocktime and realtime fail to account for clock drift in the period under consideration, and must be corrected by a factor $(1 \pm \rho)$. Although this issue is bound to become manifest during formal proof, it is worth noting that the type system of PVS detects the problem much earlier. Direct correction of the issue in the sparse timebase formulation produces unattractive results, which we finesse by adjusting the "reasonableness" condition on `GLOBALtime`. We believe that a larger overhaul of the sparse timebase design pattern could deliver further benefits. First, the sparse timebase could adopt Rushby's technique (also used in PALS) by delaying the generation and timestamping of events until all nodes are sure to be on the same global tick, thereby avoiding the need to escalate the parameter π from 0 to $1g$. Second, rather than formulating the bounds of π/Δ-precedence in terms of `realtime`, then having to deal with the fact that events are actually generated according to a schedule in `GLOBALtime`, plus the complication that clocks can drift during the relevant intervals, a formulation directly in terms of `clocktime` could yield a simpler theory and tighter bounds. We suspect that similar reparameterization of PALS would likewise simplify the statement of its corrected constraints.

PALS has a second issue (corrected in [10] and [11]) that concerns the wait between starting a round and beginning message transmission: this cannot be negative and its correction distinguishes the case where the minimum message delay is less than clock skew.

The formal verifications performed here build directly on that described in [16] (i.e., they literally import and extend the PVS theories, as corrected by Pike [14], of that previous verification), which developed a scheme equivalent to PALS, but using a different parameterization for the network properties. Modulo these differences, the corrected PALS results agree with Rushby's. A mechanically supported formal specification and verification is thus an intellectual investment that continues to provide benefit long after serving its initial purpose (we have seen the same benefit previously [15]).

The TTA sparse timebase, Rushby's scheme, and PALS all make identical assumptions (modulo choice of parameters) about minimal synchrony (i.e., bounded message delays), and clock synchronization. Hence, the choice of network architecture and design pattern should depend on pragmatics and formal considerations orthogonal to those required to support the models of computation considered here.

For the future, we plan to examine other design patterns that deliver similar services and MoCs to those considered here, but build on different network architectures: for example the Loosely Time Triggered Architecture [20] and the Globally Asynchronous Locally Synchronous (GALS) architecture [2].

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*). CoMMiCS is hosted by the Computer Science Laboratory at SRI International where this work has been performed. The second author was supported in part by NSF grant CNS-0720908 and by NASA contract NNA10DE73C. The content is solely the responsibility of the authors and does not necessarily represent the official views of the EU, NSF or NASA.

References

[1] ARINC, 2009, *ARINC Report 664P7-1 Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. AERONAUTIC RADIO, INC., 2551 Riva Road, Annapolis, Maryland 21401-7465.

[2] Chapiro, D. M. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford Univ., CA., 1984.

[3] Dolev, D., C. Dwork, and L. Stockmeyer, 1987, On the minimal synchronism needed for consensus. *Journal of the ACM*, 34(1):77–97.

[4] Fischer, M. J., N. A. Lynch, and M. S. Paterson, 1985, Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[5] Kopetz, H., 1992, Sparse time versus dense time in distributed real-time systems. In *ICDCS*, pp. 460–467.

[6] Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[7] Kopetz, H., 1998, The time-triggered model of computation. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 168, Washington, DC, USA. IEEE Computer Society.

[8] Kopetz, H. and G. Bauer, 2003, The Time-Triggered Architecture. In *Proceedings of the IEEE*, pp. 112–126. IEEE.

[9] Lynch, N. A. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996.

[10] Meseguer, J. and P. C. Ölveczky, 2010, Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In Dong, J. S. and H. Zhu (Eds.), *Twelfth International Conference on Formal Engineering Methods (ICFEM 2010)*, volume 6447 of *Lecture Notes in Computer Science*, pp. 303–320, Shanghai, China. Springer-Verlag.

[11] Miller, S., D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem, 2009, Implementing logical synchrony in integrated modular avionics. In *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, pp. 1.A.3–1 – 1.A.3–12.

[12] Owre, S., J. Rushby, N. Shankar, and F. von Henke, 1995, Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125. PVS home page: `http://pvs.csl.sri.com`.

[13] Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, 1999, *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA.

[14] Pike, L., 2006, A note on inconsistent axioms in Rushby's "Systematic formal verification for fault-tolerant time-triggered algorithms". *IEEE Transactions on Software Engineering*, 32(5):347–348.

[15] Rushby, J., 1994, A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pp. 304–313, Los Angeles, CA. Association for Computing Machinery. Also available as NASA Contractor Report 198289.

[16] Rushby, J., 1999, Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660.

[17] Rushby, J., S. Owre, and N. Shankar, 1998, Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720.

[18] Sha, L., A. Al-Nayeem, M. Sun, J. Meseguer, and P. Ölveczky. PALS: Physically Asynchronous Logically Synchronous Systems. available from: http://www.ideals.uiuc.edu/handle/2142/11897.

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

[20] Tripakis, S., C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale, 2008, Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.*, 57(10):1300–1314.

[21] Verissimo, P. and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.