

Formal Verification for Time-Triggered Clock Synchronization*

Holger Pfeifer, Detlef Schwier, Friedrich W. von Henke
Universität Ulm
Fakultät für Informatik
D-89069 Ulm

{pfeifer, schwierig, vhenke}@informatik.uni-ulm.de

Abstract

Distributed dependable real-time systems crucially depend on fault-tolerant clock synchronization. This paper reports on the formal analysis of the clock synchronization service provided as an integral feature by the Time-Triggered Protocol (TTP), a communication protocol particularly suitable for safety-critical control applications, such as in automotive “by-wire” systems. We describe the formal model extracted from the TTP specification and its formal verification, using the PVS system. Verification of the central clock synchronization properties is achieved by linking the TTP model of the synchronization algorithm to a generic derivation of the properties from abstract assumptions, essentially establishing the TTP algorithm as a concrete instance of the generic one by verifying that it satisfies the abstract assumptions. We also show how the TTP algorithm provides the clock synchronization that is required by a previously proposed general framework for verifying time-triggered algorithms.

1 Introduction

Distributed dependable real-time systems crucially depend on fault-tolerant clock synchronization. This is particularly true in distributed architectures in which processes (or nodes) perform their actions according to a pre-determined, static schedule, i.e. triggered by the progress of time. Such “time-triggered architectures” are commonly proposed or used in safety-critical applications, such as automotive control functions [5, 7]. Obviously, clock synchronization is a central element of a

*This work has been partially supported by the European Commission under the ESPRIT OMI project 23396 “Time-Triggered Architecture (TTA)”.

©1999 IEEE. Personal use of this material is permitted. However, permission to reprint or republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

time-triggered architecture for it to function properly: it is essential that the clocks of all processes be kept sufficiently close together and that the synchronization be able to tolerate faults to a limited extent.

The main purpose of this paper is to present a formal analysis of the clock synchronization algorithm embedded in a specific time-triggered context, the “Time-Triggered Protocol” (TTP) [8, 9]. TTP is the core of the communication level of the Time-Triggered Architecture, an architecture that has been developed and evaluated in two recent European projects: “Time-Triggered Architecture” (Esprit OMI program) and “X-by-Wire” (Brite EuRam program). The Time-Triggered Architecture is intended to be employed in connection with devices controlling safety-critical electronic systems without mechanical backup, so-called “by-wire” systems, e. g. for steering, braking, or suspension control [16]. Hence, high trust must be placed in its correct functioning. It has been argued that the kind of reliability required in such situations cannot be achieved without a careful formal analysis of the mechanisms and algorithms involved [2].

In the Time-Triggered Protocol, several distinct services, such as clock synchronization, group membership or redundancy management, are integrated. Among these services, clock synchronization is the most basic since for achieving the required real-time properties the other services rely on its providing a time basis common to all processors. It is for this reason that clock synchronization has been chosen as the starting point for a formal analysis of TTP. The different services provided by TTP are so tightly integrated that for the formal analysis it is first necessary to extract a clock synchronization algorithm from the integrated protocol by abstracting from those features that are irrelevant for synchronization. Furthermore, the existing description of TTP [8] is “structured English” that has to undergo a process of formalization to obtain a formal specification. The resulting formal model can then be subjected to a rigorous mathematical analysis. Distributed fault-tolerant algorithms like those for clock synchronization are inherently difficult to reason about. However, it has been observed (first by F. Schneider [17]) that the correctness arguments, i. e. the verification of the essential properties, for many synchronization algorithms are quite similar and can be derived from rather general assumptions. In our verification of TTP clock synchronization we make use of this observation and of our own formalization of the generic derivation in PVS [18]. However, the existing PVS model could not directly be used for TTP, since the synchronization algorithm of TTP differs from other algorithms in various ways: First, there are no special synchronization messages that provide the reading of a node’s clock to other nodes; instead, the delay in the arrival of incoming messages is used to estimate the value of the sender’s clock. Second, TTP provides a means to collect timing information only from selected nodes; this feature is intended for ignoring clock values of nodes that are known to have oscillators of inferior quality. Finally, the estimated readings of only four clocks are used for the calculation of a correction term of a node’s clock, even if the cluster consists of more than four nodes. Because of these peculiarities of TTP the generic PVS model for verifying clock synchronization algorithms had to be generalized. The main effort for proving the correctness of the

TTP synchronization algorithm is then to establish a mapping from the specification of the TTP algorithm to the notions that are used in the generic verification. The formal development has been carried out using the PVS system [13]; in this way, we have obtained a complete and mechanically checked formal verification of the TTP algorithm.

To put our development into a larger context, we also link it to previous work by J. Rushby (using PVS) on a general framework for verifying time-triggered algorithms; specifically, we show how the TTP algorithm provides the clock synchronization that is required by that framework. The link is facilitated by our structuring of specifications in such a manner that they “fit” the framework seamlessly. This is similar to the demonstration by Di Vito and Butler [3] that the treatment of the interactive convergence algorithm presented in [15] satisfies the synchronization requirements of the Reliable Computing Platform.

The remainder of this paper is structured as follows. The next sections give an overview of the Time-Triggered Protocol to the extent needed for this paper and describe the extraction of the clock synchronization algorithm. The formal verification presented in Section 4 consists of two parts: the first part summarizes the generic arguments, the assumptions on which they are based, and the adaptation of the generic model required for TTP; in the second part, we show, by way of example, how our formal model of TTP clock synchronization satisfies the assumptions. A subsequent section discusses the embedding into the verification framework for time-triggered algorithms. The concluding section summarizes the presented work and gives an outlook of ongoing and planned extensions.

2 Clock synchronization in the Time-Triggered Protocol

The distinguishing characteristic of time-triggered systems is that all system activities are initiated by the progress of time. From an abstract point of view, the TTP protocol operates cyclically. Each node is supplied with a clock and a static schedule, the *message descriptor list (MEDL)*. The schedule determines when certain actions have to be performed, in particular when messages of a certain type are to be sent by a particular node. The message descriptor list contains an entry which determines at which clock time a particular slot begins.

The MEDL contains global information common to all nodes in the cluster about the communication structure, such as the duration of a given slot or the identity of the sending node. As the intended system behavior is thus known to all nodes, important information can be obtained indirectly from the messages. For example, explicit acknowledgments need not be sent since a receiving node can determine that a message is missing immediately after the anticipated arrival time has passed. Similarly, from the successful reception of a message is a sufficient condition for the sending node to be considered active.

The nodes communicate via a replicated broadcast bus. Access to this bus is determined by a time-division multiple access (TDMA) schema which is pre-compiled into the schedule. Every node thus owns certain slots, in which it is

allowed to send messages on the bus. A complete cycle during which every node has had access to the bus once is called a *TDMA round*. After a TDMA round is completed, the same temporal access pattern is repeated again. The length of the message descriptor list reflects the number of different TDMA rounds and determines the duration of the so-called *cluster cycle* which, as the name suggests, is repeated over and over again.

In each slot, one of the nodes of a cluster sends a frame on each of the two channels of the bus, whereas the other nodes listen on the bus for incoming messages for a certain period of time, the *receive window*. According to certain aspects of the received message, such as content, arrival time, etc., each node then changes its internal state at some point in time before the next slot begins. Each slot is conceptually divided into two phases: during the first, the *communication phase*, the current sender is broadcasting a message via the bus; in the second, in the *computation phase*, each node changes its internal state depending on the current state and the received message. In TTP, these phases roughly correspond, respectively, to the receive window, the time frame within which nodes expect messages to arrive, and to the inter-frame gap, during which is silence on the bus.

Obviously, the clocks of the nodes must be synchronized tightly enough for them to agree on the current slot and to scan the bus at appropriate times for messages to arrive. To prevent a faulty node from speaking out of turn, the bus interface is controlled by a “bus guardian” that has independent knowledge and gives access to the bus only at appropriate times. Each node is supplied with a *physical clock* that is typically implemented by a discrete counter. The counter is incremented periodically, triggered by a crystal oscillator. As these oscillators do not resonate with a perfectly constant frequency, the clocks drift apart from real time. It is the task of clock synchronization algorithms to repeatedly compute an adjustment of a node’s physical clock in order to keep it in agreement with the other nodes’ clocks. The adjusted physical clock is what is used by a node during operation and it is commonly called a node’s *local clock*.

The general way clock synchronization algorithms operate is to gather estimates of the readings of other nodes’ clocks to estimate an adjustment for the local clock. Since every node knows beforehand at which time certain messages will be sent the difference between the time a message is expected to be received by a node and the actual arrival time can be used to calculate the deviation between the sender’s and the receiver’s clock. In this way, no special synchronization messages are needed in TTP. The time measurements are stored on a push-down stack of depth four with the most recent one on top. Thus, older values get discarded after a while. In general, there are more than four nodes in a cluster and hence not every node contributes to the calculation of a new correction term for a node’s local clock. This approach is feasible under the hypothesis that at most one of the values on the stack may be faulty in some sense, i. e. does not represent a proper clock reading.

TTP allows messages from nodes with clocks of minor quality to be excluded from the calculation of adjustments in order to improve the precision of the synchronization. This is accomplished by selecting the messages according to a *SYF*

flag (for *synchronization frame*) in the message descriptor list. If this flag is not set in the MEDL for the current slot, the obtained time difference value is not stored on the stack.

In some slots, after the communication, the adjustment term is calculated from the time values on the stack. The slots in which this is to occur are marked in the message descriptor list by a special flag named *CS* (for *clock synchronization*). TTP uses the Fault-Tolerant Average Algorithm (FTA) [10] to calculate the adjustment: The largest and the smallest value are discarded and the average of the remaining two is used as the new adjustment term.

To summarize, the TTP clock synchronization algorithm that is executed by each node individually can informally be described as follows: In every slot, perform the following steps:

1. Determine the difference between expected and observed arrival time for the incoming message.
2. If a valid message has been received and the SYF flag is set in the message descriptor list for the current slot, then push the measured time difference value onto the stack; otherwise discard it.
3. If the CS flag is set in the message descriptor list for the current slot, calculate a new correction term using the four values held on the stack and adjust the local clock accordingly.

Obviously, there are certain constraints on how the SYF and CS flags can be set in the message descriptor list for the various slots. First of all, the flags must be equally set in the message descriptor lists of all nodes. Moreover, the SYF flag must be set frequently enough in order to collect sufficiently many new time difference values. As the TTP algorithm is designed to tolerate one arbitrary (Byzantine) fault in every TDMA round, there must be at least four slots in every TDMA round with the SYF flag set.

3 Formal model for the TTP synchronization algorithm

This section describes a formal specification of the TTP clock synchronization algorithm that has been developed from an informal description of the TTP protocol [8]. Since in TTP many different services are tightly integrated the first step towards a formal model is to abstract from those features that are not relevant for clock synchronization in order to make the mechanical analysis feasible at all. In particular, the internal state of a node which in the TTP implementation comprises quite a number of registers for various purposes has been reduced in the model to contain only a few components. More precisely, the internal state of a node is modeled as a record consisting of the following elements:

- a counter *current_slot* which records the number of the current slot,

- a stack *timediffs* of depth four for storing the time difference values,
- two registers *current_correction* and *total_correction* that contain the value of the most recently calculated clock adjustment and the sum of all adjustments calculated so far, respectively.

In addition we assume another stack, also of depth four, to record the slots in which the corresponding entries of the stack of time values have been obtained. This stack is not a component of the TTP data structure, but it is needed for verifying the synchronization algorithm.

We use the PVS notation of projection functions to denote the components of the state record; thus *timediffs(s)*, for example, denotes the stack of time difference values of state *s*.

Initially, the slot counter, the adjustment values, and the entries of the stack of time difference values are all set to zero; the stack of slot numbers is initialized with negative values in order to distinguish them from proper slot numbers; the latter are represented by natural numbers. The function *initialstate* maps every node to its initial state.

Usually two views of time are distinguished: real time is given by some external frame of reference while clock time is a node's local approximation provided by the physical clock. Real time is taken as ranging over the real numbers, and integer values are used to model clock time. We adopt the convention of using lower-case variables, such as *t* or t_p^i to denote real-time entities whereas clock-time quantities will be denoted by upper-case identifiers. In our formalization the physical clock of a node *p* is modeled by a function PC_p which maps real time to clock time; thus, $PC_p(t)$ denotes the reading of *p*'s physical clock at real time *t*. The rate at which a physical clock may drift apart from real time is assumed to be bounded by a small positive quantity.

The reading of the local clock of a node *p* in some given state *s* at real time *t* is obtained by adding the adjustment adj_s to the reading of the node's physical clock PC .

$$\begin{aligned} adj_s &= current_correction(s) + total_correction(s) \\ LC_p^s(t) &= PC_p(t) + adj_s \end{aligned}$$

In order to describe the state of a node at particular clock times we introduce a function *schedule(r)* which denotes the clock time at which a given slot *r* starts. The schedule is not directly available in TTP, but there is an entry for the duration of each slot *r* in the message descriptor list; this is formally captured by the function *duration*. Given a clock time constant *system_start_time* which is assumed to initially show up on every node's local clock, it is a simple matter to define *schedule* by recursively summing up the duration of the slots:

$$schedule(r) = \begin{cases} system_start_time & \text{if } r = 0 \\ schedule(r-1) + duration(r-1) & \text{if } r > 0 \end{cases}$$

The state of a node p at a certain clock time T is given by a function $ttss$ (for *time-triggered system state*). The definitions involving $ttss$ are adapted from work by Rushby [14]. By the start of the first slot, p is in its initial state:

$$ttss(p)(\text{schedule}(0)) = \text{initialstate}(p)$$

Let $\text{comm_duration}(r)$ denote the duration of the communication phase of a slot r , during which a node p waits for a message to arrive. Within the communication phase the internal state of p remains unchanged:

$$\begin{aligned} ttss(p)(T) &= ttss(p)(\text{schedule}(r)) \\ &\text{for } \text{schedule}(r) \leq T \leq \text{schedule}(r) + \text{comm_duration}(r) \end{aligned}$$

At some point during the computation phase node p is changing its internal state depending on its current state and the message it has received. This behavior is described by a state transition function trans such that $\text{trans}(p, m)(s)$ denotes the next state of a node p that has received message m in state s . The state of p is unspecified during the computation phase; all that is said is that by the beginning of the next slot $\text{schedule}(r + 1)$ node p has changed into a new state.

$$\begin{aligned} ttss(p)(\text{schedule}(r + 1)) &= \text{trans}(p, \text{tin}(p, T))(ttss(p)(T)) \\ &\text{where } T = \text{schedule}(r) + \text{comm_duration}(r) \end{aligned}$$

Here, the function $\text{tin}(p, T)$ models the message p receives at clock time T . As the contents of the message is irrelevant as far as clock synchronization is concerned (only the arrival times of messages are of importance) we need not to be too specific about the reception of messages and leave tin uninterpreted. For our purpose it is sufficient to assume a set *Message* of messages with a distinguished element *null* to model the case when a node has not received any message.

So far we have described the general behavior of a node in a time-triggered system. What remains is to model the state transitions that a node performs in each slot, i. e. the state transition function trans . This function formalizes the clock synchronization algorithm that has informally been described in the previous section. From the synchronization point of view there are two sorts of slots: “ordinary” ones in which only message delays are measured, and slots in which a new correction term is calculated. Accordingly, the definition of the state transition function trans is divided into two parts: the first is described by the function $\text{do_slot}(p, m, s)$ which is evaluated in every slot and gathers the time difference values and stores them on the stack. The second function, $\text{update}(p, s)$, computes new values for the correction terms. The latter function is only evaluated if in the current slot the clock synchronization algorithm is to be executed, i. e. when the CS flag is set in the message descriptor list for the current slot.

$$\text{trans}(p, m)(s) = \text{update}(p, \text{do_slot}(p, m, s))$$

The deviation of the arrival time of a message is measured by a hardware mechanism that is captured by the function $\text{get_time_diff}(p, m, s)$ in the formal model.

Time difference values are only stored on the stack if the received message is *valid*. This is the case, e. g. if, among other things, it has been received within the receive window. In particular, *null* denoting that no message has been received is not valid. We use a predicate $new_timediff_available?(p, m, s)$ to model whether a time difference is available for being stored on p 's stack. In this case, function $do_slot(p, m, s)$ pushes the current time difference value onto the stack; in addition, the current slot number is pushed onto the second stack to record the origin of each time measurement. Finally, the slot counter is increased.

$$\begin{aligned}
 do_slot(p, m, s) &= s', \text{ where} \\
 current_slot(s') &:= current_slot(s) + 1 \\
 \text{and if } new_timediff_available?(p, m, s) &\text{ then:} \\
 timediffs(s') &:= push(get_time_diff(p, m, s), timediffs(s)) \\
 slots(s') &:= push(current_slot(s), slots(s))
 \end{aligned}$$

If the clock synchronization algorithm is to be executed in the current slot the function *update* updates the values of the correction terms according to the result of the fault-tolerant average algorithm. The latter is specified by the function *calculate_correction* that takes the stack of time difference values as its argument. The predicate *syncround?* is true if the CS flag is set in the message descriptor list for the current slot.

$$\begin{aligned}
 update(p, s) &= \begin{cases} s' & \text{if } syncround?(current_slot(s)) \\ s & \text{otherwise} \end{cases} \\
 \text{where } current_correction(s') &:= calculate_correction(timediffs(s)) \\
 total_correction(s') &:= current_correction(s) \\
 &\quad + total_correction(s)
 \end{aligned}$$

4 Verification of the TTP synchronization algorithm

Clock synchronization algorithms do not only have the task of keeping the clocks of a cluster of nodes tightly together. As with distributed algorithms in general, clock synchronization is usually required to work also in the presence of faults. Algorithms differ in the number and kind of faults they are designed to tolerate. The TTP algorithm, for example, is able to handle one asymmetric (Byzantine) fault in each TDMA round, i. e., as long as at most one fault occurs in any consecutive n slots, where n is the length of a TDMA round, the algorithm is able to maintain the clocks synchronized.

The required fault-tolerance of an algorithm makes it inherently difficult to reason about it, since careful attention has to be drawn to faults and failed components. Schneider [17] has observed, however, that the correctness arguments of so-called *averaging algorithms* are quite similar. This class of algorithms is described using a *convergence function*. Schneider stated several rather general assumptions on the convergence function and showed that they are sufficient to prove the correctness of the algorithm. Subsequently, Shankar used the EHDM system to mechanically

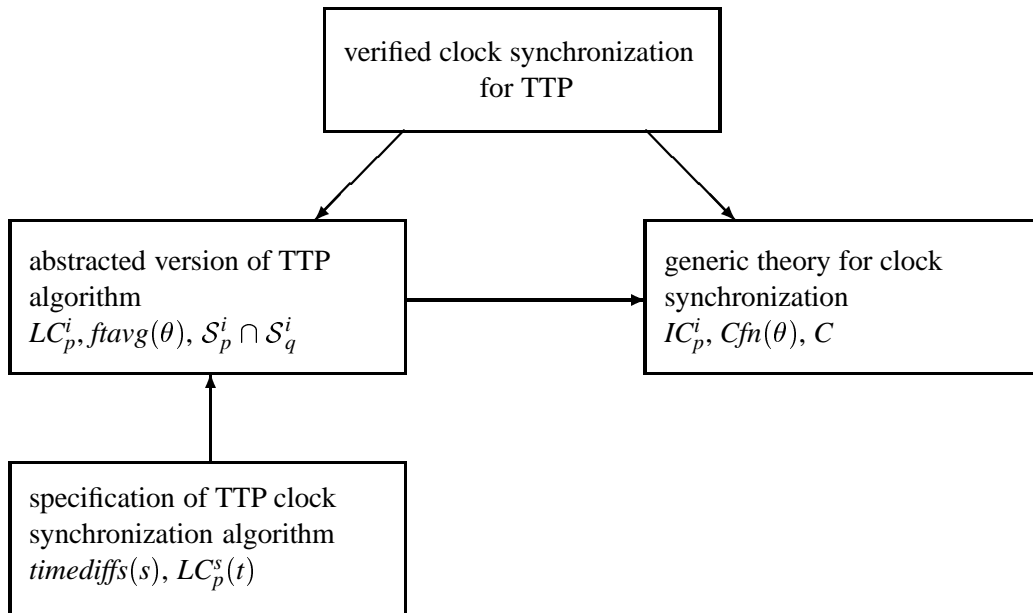


Figure 1: Structure of the TTP instance of the generic theory for clock synchronization.

verify Schneider’s proof [19, 20] and Miner [12], and more recently Schwier and v. Henke [18] have further improved the constraints and the organization of the proof itself, respectively.

By following this general two-step approach to clock synchronization we can make use of existing PVS formalizations of the generic synchronization proof. This reduces the reasoning effort required to complete the proof compared to a verification directly from the low-level specification presented in Section 3. The argument for the correctness of the synchronization is first derived from a set of generic assumptions that are independent from a particular algorithm. The second step is then to show that the assumptions on which the clock synchronization proof is based are indeed satisfied by the concrete TTP algorithm.

For the verification of the TTP clock synchronization we utilize a variant of our own formalization of the generic derivation in PVS [18]. This formalization is similar to Miner’s development [12] in EHDM, a predecessor of the PVS verification system, with the organization of the various PVS theories and proofs being improved to also incorporate non-averaging algorithms. The existing formalization, however, had to be generalized in order to accommodate it to the particular needs of TTP. This generalization involved the modification of the signatures of some of the parameters, in particular that of the convergence function.

The remainder of this section first briefly summarizes the generic model for verifying clock synchronization algorithms. It is, however, beyond the scope of this paper to restate in detail the proof of the synchronization property and all of the assumptions this proof is based on. Instead, we take one of the central conditions

on the convergence function as an example and describe the generalization that was necessary to capture the peculiarities of the TTP algorithm. Then we explain how the generic model is used to yield a correctness proof for the TTP synchronization algorithm. The major task in deriving a TTP instance of the generic theory is to gradually abstract the specification of the synchronization algorithm presented in the previous section to the level of the generic model and to define an appropriate translation between those two abstractions. The overall structure of this development is depicted in Figure 1.

4.1 Generic model for verifying clock synchronization

A clock synchronization protocol implements a *virtual clock*¹ by repeatedly adjusting a node's physical clock. The task of the synchronization algorithm is to bound the *skew*, i. e. the absolute difference between the virtual clock readings of any two (non-faulty) nodes p and q by a small value δ at any time t :

$$| VC_p(t) - VC_q(t) | \leq \delta$$

This property is commonly called *agreement* and relates the readings of two clocks. The other important property, *accuracy*, is concerned with the quality of the approximation to real time by a clock; this is, however, not discussed in this paper.

The proof of this property is generally accomplished through mathematical induction on the number of synchronization intervals. The induction hypothesis states that at the beginning of each interval, the skew between any two clocks is bounded by some value $\delta_S < \delta$. Then it is shown that during the next interval, when the clock readings drift apart, the skew does not exceed δ . Finally one has to prove that the application of the convergence function brings the clocks together again within δ_S . The latter step is the harder one, since the former rather imposes certain constraints on the maximum precision that can be achieved given concrete values for the drift rate ρ of the clocks and the length of a synchronization interval.

To facilitate the induction proof several additional concepts and notations have proven useful, in particular the abstract notion of *interval clocks*. Instead of repeatedly applying adjustments to a local clock one could also think of a node starting a new clock each time the synchronization algorithm has been executed. These clocks are indexed by the number of the synchronization interval i and are denoted $IC_p^i(t)$. The value of p 's interval clock in the i th synchronization interval is obtained by adding the i th adjustment to the reading of p 's physical clock:

$$IC_p^i(t) = PC_p(t) + adj_p^i$$

These interval clocks are then put together to form the node's virtual clock: in the i th synchronization interval p 's virtual clock corresponds to the i th interval clock:

$$VC_p(t) = IC_p^i(t), \text{ for } t_p^i \leq t < t_p^{i+1}$$

¹In the description of the TTP algorithm the virtual clocks have been called *local clocks*.

Here, t_p^i denotes the begin of the i th synchronization interval. The way the adjustments to a node's physical clock are computed is abstractly captured by the concept of a *convergence function* Cfn . The convergence function takes an array Θ_p^i of readings of the clocks of some or all other nodes to calculate a corrected clock reading for p . The value $\Theta_p^i(q)$ is p 's estimate of q 's clock reading at time t_p^i . The adjustment to p 's physical clock is then given by the difference of its physical clock and the result of the convergence function; initially it is taken to be 0:

$$\begin{aligned} adj_p^0 &= 0 \\ adj_p^{i+1} &= Cfn(p, \Theta_p^{i+1}) - PC_p(t_p^{i+1}) \end{aligned}$$

Schneider has stated several conditions that are necessary to complete the proof of the bounded skew property. Some of them, e. g. those concerning the interrelationships among the various quantities introduced, are of minor importance in that they can be derived more easily for concrete algorithms. The most important of the conditions are concerned with the behavior of the convergence function that a clock synchronization algorithm exploits. The usefulness of these conditions is for the most part due to its isolation of purely mathematical properties from other concepts such as, e. g., failed nodes. We consider one of them, called *precision enhancement*, in more detail.

The property *precision enhancement* is used to bound the skew between two clocks immediately after the application of the convergence function. The actual bound depends on the skews between the value in the array of estimated clock readings. Given two such arrays γ and θ used by two nodes p and q , respectively, precision enhancement states that the absolute values of the convergence function applied by p and q , respectively, do not differ by more than a quantity $\Pi(X, Y)$, provided that corresponding entries in γ and θ differ by no more than X and the values in γ and θ , respectively, fall within a range Y . Furthermore, it is required that $\Pi(X, Y) < Y$ for the precision to be truly enhanced.

There is a bound $\Pi(X, Y)$ such that
 if for all $l \in C$: $|\gamma(l) - \theta(l)| \leq X$
 and for all $l, m \in C$: $|\gamma(l) - \gamma(m)| \leq Y$ and
 $|\theta(l) - \theta(m)| \leq Y$
 then $|Cfn(p, \gamma) - Cfn(q, \theta)| \leq \Pi(X, Y)$

This formalization of the precision enhancement property is taken from Schwier and v. Henke's work [18], except for some minor notational differences. Note the use of the set C : in previous presentations of this property the convergence function is assumed to use an array, θ say, of N clock readings, where N is the number of nodes. The preconditions of precision enhancement are required to be satisfied by at least $N - F$ of these readings, with F being the number of faults to be tolerated by the algorithm; this set of readings is denoted by C . For the algorithm to tolerate any arbitrary (Byzantine) fault it is crucial that N is at least $3F + 1$ (cf. [4]). This ensures that the sets of readings used in the convergence function by two nodes overlap.

The intended interpretation of C is the set of readings from non-faulty clocks. This view is due to the implicit assumption that the array of clock readings is a mapping from nodes to clock time values. However, as we show, this is not necessarily required. Moreover, the TTP algorithm does not allow for such an interpretation of the array of readings. Of course one could use the senders of the messages that lead to the time difference values on a node's stack as the domain of θ but the problem arises with the interpretation of faulty readings: in the TTP protocol the reception of a valid message at a node q is a sufficient condition for q to consider the sender of the message to be correct. Thus, a new time difference value will only be stored on the stack if being received from a non-faulty node and communication faults result in the lack of such a new value. Therefore the problem with the readings is not that they come from some faulty node but rather that some of them might remain on the stack for "too long" if they do not get pushed out by new values. Thus these old values may not represent accurate estimates of the remote clock readings; in the worst case they haven't been even gathered in the most recent synchronization interval.

For the verification of the TTP algorithm it was therefore necessary to generalize the treatment of the array of clock readings and the actual form of the generic assumptions that involve the convergence function. In particular, we allow the domain of the array of clock readings to be any set of size N with different nodes possibly having different sets and we define C to be the intersection of the respective sets of two nodes that is required to contain at least $N - F$ elements for any two nodes.

The two other of Schneider's constraints on the convergence function, called *accuracy preservation* and *translation invariance*, are affected by this generalization, too. They are, however, omitted in this presentation. For a detailed explanation of these and the other conditions we refer to Schneider [17] and Miner [12]; a complete generic derivation of the synchronization property from these conditions is given by Schneider [17] and Shankar [19].

4.2 Deriving abstract properties of the protocol

While the formal model of TTP is describing the clock synchronization algorithm on the level of slots, the generic verification is based on the notation of synchronization intervals. In order to exploit the generic proof of clock synchronization for the TTP algorithm the concrete model of TTP has to be abstracted to the level of the concepts used in the generic model. This means in particular that the definition of the local clocks and the calculation of the adjustments needs to be in terms of interval clocks and a convergence function.

A first step towards this goal is to derive from the slot-based description of local clocks an interval-based one. Obviously the adjustment adj_s is only changed if the synchronization algorithm is executed in the current slot. The slots in which this is to occur are marked in the message descriptor list with the CS flag set. Given a function *syncround* such that *syncround*(i) yields the number of the slot in which the CS flag is set for the i th time we can define the i th adjustment of p , denoted

adj_p^i , as the adjustment adj_s given in p 's state after the synchronization algorithm has been invoked for the i th time. Similarly, an interval-based description of the local clock, denoted LC_p^i , is defined:

$$\begin{aligned} adj_p^i &= adj_p^s \\ LC_p^i(t) &= LC_p^s(t) \\ \text{where } s &= \begin{cases} \text{initialstate}(p) & \text{if } i = 0 \\ \text{tss}(p)(\text{schedule}(\text{syncround}(i) + 1)) & \text{if } i > 0 \end{cases} \end{aligned}$$

It is easy to see that the following equation holds for $LC_p^i(t)$:

$$LC_p^i(t) = PC_p(t) + adj_p^i$$

In previous work on clock synchronization clocks are also sometimes expressed in terms of functions mapping clock time to real time [11, 12, 15]. Some of our definitions and proofs are more naturally described this way and we therefore introduce the inverse mapping pc_p of p 's physical clock; $pc_p(T)$ denote the earliest real time that p 's physical clock reads T . Thus, we can define an inverse mapping of LC_p^i as

$$lc_p^i(T) = pc_p(T - adj_p^i)$$

In order to cast LC_p^i into the form used in the definition of the interval clock IC_p^i several additional notations have to be introduced. First, we define the real time instant t_p^i at which p invokes the synchronization algorithm for the i th time by means of lc_p^i . Here, the clock time $\text{scheduled_synctime}(i)$ denotes some instant in the computation phase of the i th synchronization slot at which the synchronization algorithm is executed.

$$t_p^i = \begin{cases} lc_p^0(\text{system_start_time}) & \text{if } i = 0 \\ lc_p^{i-1}(\text{scheduled_synctime}(i)) & \text{if } i > 0 \end{cases}$$

The next step is to formulate the adjustments adj_p^i in terms of a convergence function. First, the array of clock readings Θ_p^i has to be defined. As described above, each node p maintains a stack of time difference values. These values are used to calculate an estimate of the reading of a remote clock by adding the time difference to the value of p 's local clock at time t_p^i . As explained in the previous subsection, the array of readings can not be modeled by a function mapping nodes to clock readings. Under certain conditions it can even occur that there are two values from the same sender on the stack². Therefore it is not the sender of a message but the slot number in which the message was sent that is the domain of the function Θ_p^i . The slot numbers are recorded separately on a additional stack. Note again that this stack is only an abstract concept that is used for the verification but is not implemented in the protocol.

²This might be the case in a cluster of four nodes when a communication fault occurs in the last TDMA-round before the synchronization.

We use \mathcal{S}_p^i to denote the set of slot numbers that are contained on p 's stack in the i th synchronization interval. Moreover, idx_p^i is a mapping that yields for every slot number s the index on p 's stack at which s is stored. The elements of the stack are denoted $stack.0$ (top) to $stack.3$ (bottom). The array of clock readings Θ_p^i is then modeled as a function mapping the values of the stack of slots to the corresponding entries of the stack of time difference values:

$$\begin{aligned} \mathcal{S}_p^i &= \{s : \mathbb{N} \mid s = stack.0 \vee s = stack.1 \vee s = stack.2 \vee s = stack.3\} \\ \Theta_p^i &= \lambda s \in \mathcal{S}_p^i. LC_p^{i-1}(t_p^i) + stack.idx_p^i(s) \\ &\text{where } stack = timediffs(ttss(p)(schedule(syncround(i) + 1))) \end{aligned}$$

TTP uses the fault-tolerant average algorithm [10] to calculate the adjustments. In general, the algorithm takes N clock readings among which up to F readings might be faulty in some sense. The readings are sorted and the F largest and the F smallest values are discarded. The algorithm then returns the average of the remaining $N - 2 * F$ readings as its result. In the case of TTP, each node has $N = 4$ readings to calculate the adjustment and it is assumed that at most one of them does not represent a proper time difference value, i. e. $F = 1$.

The formalization of the fault-tolerant average algorithm $ftavg$ assumes a function for sorting an array of readings θ that can be used to find the second largest and second smallest element, denoted $\theta_{(1)}$ and $\theta_{(2)}$, respectively.

$$ftavg(\theta) = \left\lfloor \frac{\theta_{(1)} + \theta_{(2)}}{2} \right\rfloor$$

$$Cfn(p, \Theta_p^i) = ftavg(\Theta_p^i)$$

Now we have collected all the ingredients to define the interval clocks IC_p^i :

$$\begin{aligned} adj_p^i &= \begin{cases} 0 & \text{if } i = 0 \\ Cfn(p, \Theta_p^i) - PC_p(t_p^i) & \text{if } i > 0 \end{cases} \\ IC_p^i(t) &= PC_p(t) + adj_p^i \end{aligned}$$

Despite the various additional notations the interval clocks are nothing but an abstracted version of the local clocks introduced in the previous section. In fact, one can prove the following theorem that relates interval clocks to local clocks:

$$\text{For all } p, i, \text{ and } t: IC_p^i(t) = LC_p^i(t)$$

For the rest of this section we briefly sketch the derivation of the precision enhancement property described in the previous subsection for the TTP convergence function Cfn . The formalized proof follows closely the one presented by Miner [12] for the fault-tolerant midpoint algorithm which coincides with the TTP algorithm since only two values are used for averaging. For both of these convergence functions, the bound $\Pi(X, Y)$ is given by

$$\Pi(X, Y) = \left\lceil X + \frac{Y}{2} \right\rceil$$

The crucial step in the proof of precision enhancement is to show that for any two nodes there is at least one good reading in the range of values that are selected for the computation of the average by those nodes; this is more formally stated in the following lemma:

Given two arrays of readings θ and γ , there exists a $l \in C$ such that

$$\gamma(l) \leq \gamma_{(1)} \quad \text{and} \quad \theta_{(2)} \leq \theta(l)$$

For the TTP instance we define C as the intersection of the domains \mathcal{S}_p^i and \mathcal{S}_q^i of the two readings θ and γ , respectively. In order to accomplish the proof of this lemma, C has to contain at least $N - F$ elements and N must be greater or equal $3F + 1$. While the latter constraint is trivially true for TTP as N equals 4 and F is 1, the former requires more effort to be validated. In the concrete TTP instance this constraint requires us to show that the intersection of the slot numbers on the stacks of any two nodes p and q contains at least 3 elements. The derivation of this property can informally be described as follows:

Case 1: The messages that are sent in the last four slots immediately before the invocation of the synchronization algorithm are received correctly by both p and q .

Hence, both nodes have the same slot numbers on their stacks and thus the size of C is 4.

Case 2: A fault occurred in the last four slots immediately before the invocation of the synchronization algorithm.

In this case, one of the two nodes has received a valid message, while the other has not. In TTP it is assumed that at most one such fault occurs in any n consecutive slots where n is the length of a TDMA round. For the TTP algorithm to tolerate a Byzantine fault n must be greater or equal 4. If less than 4 nodes are left in the network, the Byzantine requirement is waived for TTP [8]. In the case of a fault one of the two nodes stores a new time difference value and the corresponding slot number, x say, on its stacks while the other does not. At this time, C would contain 3 elements. The size of the set of common slots can be further decreased only if another fault occurs before the “bad” value x is pushed out of the stack, that is, within the next three slots. This is, however, contrary to the hypothesis that faults occur at least 4 slots apart. Hence, any two given nodes have at least three time difference values from the same set of slots on their stacks at the time the synchronization algorithm is executed.

The formal verification of this property in PVS turned out to be quite challenging, especially because the additional feature of discarding correct messages from some nodes according to the *SYF* flag had to be taken into account, too. This required some subtle reasoning about the cardinality of various sets of slot numbers. The complete PVS formalization contains quite a number of the definitions and proved formulas of which only the fewest can be described in this paper.

5 Integration into a general framework for time-triggered systems

As in the context of general program verification it is a natural approach to verify the various aspects of fault-tolerant algorithms at different levels of abstraction that capture the essence of the property under concern. Following this idea of a hierarchical treatment J. Rushby has presented a framework for a systematic formal verification of time-triggered implementations of round-based algorithms [14].

The algorithm is first specified as a functional program – a form that is best suited for a formal and mechanical analysis since at this level the proofs are generally accomplished by (more or less) simple inductions. Then the functional program is transformed into an untimed synchronous system. Although this transformation can be carried out systematically to some extent [1, 14], the correctness of this step must be accomplished separately. The last step is then to refine the untimed system into a time-triggered implementation. The correctness of the latter step can be verified independently of the algorithm concerned. Thus, provided care is taken with respect to fault modes, properties and the correctness of the algorithm directly carry over from the untimed system to the time-triggered implementation.

For the proof of the correctness of this latter transformation it is required, however, that the clocks of the nodes in the cluster are synchronized. The state of a node p in the untimed synchronous system model after a given number r of rounds is specified by a function $run(r)$ that applies the state transition function $trans$ to the current state of p by recursing on r . While in the untimed system all nodes proceed in discrete steps one has to find a certain instant where the nodes of the time-triggered system all are in the same round in order to relate the global state of the time-triggered system to the one of the untimed model. Rushby defines the *global start time* of a round r , denoted $gs(r)$, to be the real time when the slowest clock begins this round and proves by establishing a simulation relationship that for all rounds the states of the two systems correspond:

$$tts(p)(VC_p(gs(r))) = run(r)(p)$$

Synchronization of the clocks is now required to ensure that faster clocks do not drift too far that some other node would have already started its computation phase (and possibly changed its state).

In order to provide the necessary synchronization we have incorporated our development for the TTP clock synchronization into Rushby's model. This required some re-organization of the PVS theories, but the overall structure of the proofs needed not to be changed. Figure 2 shows the structure of the extended model. The two boxes on the top represent a fault-tolerant algorithm specified as a functional program and expressed in an untimed synchronous system, respectively. The dashed arrow between these boxes indicates that the relationship between these two representations of the algorithm must be established by a separate correctness proof. The box at the bottom stands for the time-triggered implementation of the algorithm that can generically be shown to be a refinement of the untimed system, hence the use of a solid arrow here.

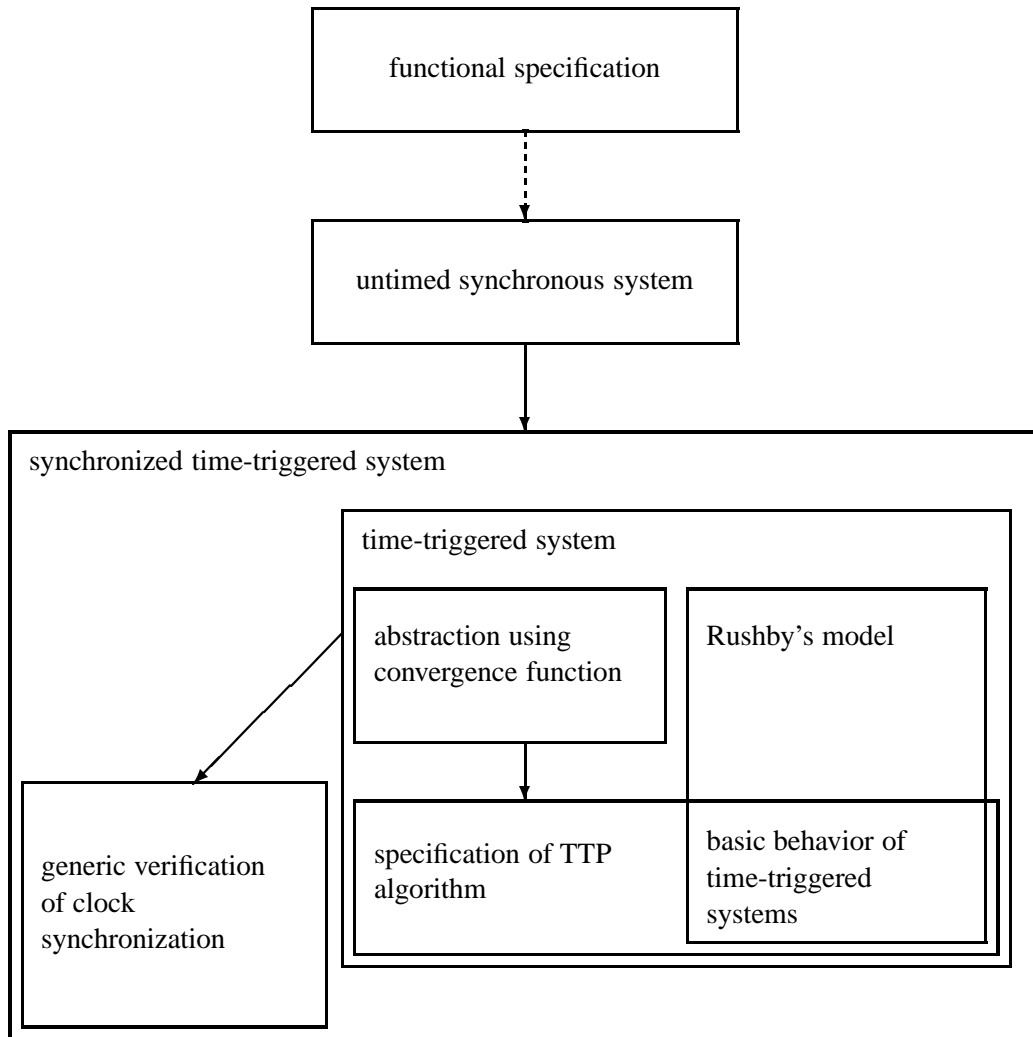


Figure 2: Structure of the general model for verifying time-triggered algorithms extended by the proof of clock synchronization.

Linking the clock synchronization proof to this general framework gives more structure to the formal model of time-triggered systems, illustrated by the box at the bottom: the right-hand side of it represents what is our adaption of Rushby's PVS theory. Some of definitions concerning the description of the basic behavior of time-triggered systems have been separated out to be used in the specification of the clock synchronization algorithm of TTP, cf. Sect. 3. The derivation of abstract properties of the algorithm together with the remaining definitions of Rushby's model form the time-triggered implementation of the TTP clock synchronization algorithm that makes use of the generic derivation to verify the synchronization property.

In this framework it is now possible to specify other services of the TTP protocol such as group membership on the level of untimed synchronous systems (cf. [6]). This more abstract level is justified by the existence of synchronized clocks. Link-

ing the clock synchronization proof to this level of abstraction thus makes the inter-relationship between various protocol services more explicit.

6 Conclusion

We have presented the formal verification of the clock synchronization algorithm that is implemented in the Time-Triggered Protocol (TTP). We have developed a formal model of TTP in the verification systems PVS. In order to make the mechanical analysis feasible the model abstracts from all features that are not relevant for clock synchronization.

For the actual verification, major emphasis has been given to making use of previous work on formally verifying clock synchronization algorithms. This led to splitting up the proof into a generic part in which the synchronization property is proved based on several abstract assumptions, and a TTP-specific part in which the specification of the algorithm is shown to satisfy those assumptions. This two-step approach reduced the overall verification effort, even though the existing generic proofs needed to be adapted and generalized to accommodate the particular needs of TTP. The specification and verification system PVS that has been used as mechanical proof assistant directly supports such an approach: theorems can be based on assumptions, and when using concrete instances of those theorems PVS serves as a book keeper that requires proofs for the concrete values to satisfy all the assumptions.

For the generic part of the verification with PVS we used a variant of the theory of Schwier and v. Henke [18] for averaging algorithms; for TTP we had to generalize some of the abstract parameters of the model and related assumptions. In the course of proving the validity of the generic assumptions for the TTP algorithm a major task has been to abstract the algorithm specification that has been developed from the existing informal TTP specification to a level at which it is expressed in terms of the concepts that are used in the generic proofs. In contrast to previous work on clock synchronization, the fault hypothesis of TTP would not be captured appropriately if faults were directly related to certain nodes. Instead of considering faulty nodes, it was therefore necessary to reason about whether or not a fault occurred in a given slot; this made it quite challenging to verify that at the times of synchronization all the nodes have readings that originate from a sufficiently large common set of slots.

The verified clock synchronization theory has been linked to Rushby's general approach to verifying time-triggered algorithms [14]. This framework assumes clocks to be synchronized; we have shown how the clock synchronization of TTP can be integrated, within a common PVS context, with the framework to provide the assumed service. This allows other protocol services to be analyzed at the level of untimed synchronous systems, rather than at the level of their time-triggered implementation, without losing the completeness in the chain of formal argumentation.

Future work will be concerned with extending the formal model to include other aspects and services of TTP. Currently, we are examining how the correctness proof

for a group membership protocol similar to the one of TTP [6] can be adapted to the actual TTP algorithm. Moreover, the general framework could be expanded to capture initialization and re-integration of nodes.

References

- [1] W. Bevier and W. Young. The Design and Proof of Correctness of a Fault-Tolerant Circuit. In J. Meyer and R. Schlichting, editors, *Dependable Computing for Critical Applications*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, pages 243–260. Springer-Verlag, 1991.
- [2] R. W. Butler and G. B. Finelli. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Trans. on Software Engineering*, 19(1):3–12, Jan. 1993.
- [3] B. Di Vito and R. Butler. Formal Techniques for Synchronized Fault-Tolerant Systems. In *Dependable Computing for Critical Applications – 3*, Dependable Computing and Fault-Tolerant Systems, pages 279–306. pringer Verlag, 1992.
- [4] D. Dolev, J. Halpern, and H. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer and System Sciences*, 36(2):230–250, April 1986.
- [5] G. Heiner and T. Thurner. Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems. In *Proc. 28th International Symposium on Fault-Tolerant Computing (FTCS '98)*. IEEE Computer Society, 1998.
- [6] S. Katz, P. Lincoln, and J. Rushby. Low-Overhead Time-Triggered Group Membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 155–169. Springer Verlag, September 1997.
- [7] H. Kopetz. The Time-Triggered Approach to Real-Time System Design. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*. Springer, 1995.
- [8] H. Kopetz. Specification of the Basic TTP/C Protocol. Internal project document, not publicly available, 1998.
- [9] H. Kopetz and G. Grünsteidl. TTP – A Time Triggered Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, January 1994.
- [10] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. Computers*, 36(8):933–940, August 1987.

- [11] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *JACM*, 32(1):52–78, Jan. 1985.
- [12] P. S. Miner. Verification of Fault-Tolerant Clock Synchronization Systems. NASA Technical Paper 3349, NASA Langley Research Center, January 1994.
- [13] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Trans. on Software Engineering*, 21(2):107–125, February 1995.
- [14] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. In M. Dal Cin, C. Meadows, and W. H. Sanders, editors, *Dependable Computing for Critical Applications – 6*, pages 203–222. IEEE Computer Society, March 1997.
- [15] J. Rushby and F. von Henke. Formal Verification of Algorithms for Critical Systems. *IEEE Trans. on Software Engineering*, 19(1):13–23, January 1993.
- [16] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture. In Jean-Yves Roger, Brian Stanford-Smith, and Paul T. Kidd, editors, *Advances in Information Technologies: The Business Challenge. Proceedings of EMMSEC'97 - European Multimedia, Microprocessor Systems and Electronic Commerce*. IOS Press, 1997.
- [17] F. B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical Report 87-859, Cornell University, Aug. 1987.
- [18] D. Schwier and F. W. von Henke. Mechanical Verification of Clock Synchronization Algorithms. In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1486 in LNCS, pages 262–271. Springer, September 1998.
- [19] N. Shankar. Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm. Technical Report CR-4386, NASA, 1991.
- [20] N. Shankar. Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer-Verlag, January 1992.