Appears in: *Proc. of the 8th IEEE Int. Conf. on Emerging Technologies and Factory Automation, ETFA 2001, pp. 343–352, Oct. 2001, IEEE.*

Formal Analysis for Dependability Properties: the Time-Triggered Architecture Example

Holger Pfeifer, Friedrich v. Henke
Abteilung Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm, D-89069 Ulm
Germany

***Abstract* – This paper describes the mechanized formal verification we have performed on some of the crucial algorithms used in the Time-Triggered Architecture (TTA) for safety-critical distributed control. We outline the approach taken to formally analyse the clock synchronization algorithm and the group membership service of TTA, summarize our experience and describe remaining challenges.**

I. INTRODUCTION

In recent years the use of computing elements to control technical systems has become more and more common. Such *embedded systems* often provide increased functionality and flexibility in comparison with the technical control systems they are replacing. In safety-critical applications, such as the control of processing plants, automotive control or avionics, the dependability of the computing element becomes a prime issue since failure may have catastrophic consequences. Embedded systems of this kind typically involve distributed, real-time computations and requirements of fault tolerance; this makes their analysis and the verification that they behave as required inherently difficult. To attain the desired – or required – level of confidence in correct behaviour, mere testing is usually insufficient; in fact, it has been argued [2,15] that the kind of reliability required for highly safety-critical applications cannot be achieved without a careful formal analysis of the mechanisms and algorithms involved.

The formal analysis of a system may be viewed as typically involving the following steps:

1. Construction of a precise, *formal description* of the mechanisms and algorithms. The description focuses on essential aspects and abstracts from inessential details. Developing such a formal model in itself often has the benefit of clarifying issues left vague or open or omitted altogether in more informal descriptions that serve as the starting point.

   The formal model is intended to be processed by machine; hence it is expressed in the formal language of the verification tool to be employed in the next step. Similarly, the properties to be verified must be expressed in a related formal language.

2. *Verification* that the system, as represented by the formal model, satisfies the stated requirements. Among verification techniques, we may distinguish two classes: model checking and theorem proving. *Model checking* [3] involves modeling the system as a finite state transition system and expressing the desired properties as formulas in a temporal logic. Model checking is the approach of choice for many embedded systems that fit the underlying modeling paradigm; it is popular because once a model has been constructed the actual analysis process is "automatic", which contributed to widespread adoption of the approach by industry. Furthermore, the approach links up well with common design methods, such as UML, that employ some form of state charts; tools for mapping state charts into suitable input for model checkers are under development.

   However, the type of analysis we will be discussing in the following section cannot be carried out with model checking as it requires, for instance, arithmetic computations for time bounds. This kind of problems can be dealt with by theorem proving: the relevant system aspects are modeled as theories in (first- or higher-order) predicate logic, and the required properties are derived as theorems by an interactive process of deduction. The two approaches to formal analysis and verification should be regarded as complementary.

In most realistic situations, an iteration of the two steps (modeling and verification) will be required to get both the formal model and the formal arguments supporting the verification right. Here, the support by a computer-based tool is crucial: the computer is much more reliable in handling the typically large amount of uninteresting details. Furthermore, a machine-supported verification can be repeated, adapted to small system changes and may eventually serve as supporting evidence in a certification of safety-critical components. In this paper we discuss the formal analysis of safety-critical properties of one particular application, the Time-Triggered Architecture, that we have performed using the specification and verification system PVS [12].

The remainder of the paper is organized as follows: after giving a brief overview of PVS in the next section, Sect. III. describes the formal analysis of two important services of

the Time-Triggered Architecture, namely the clock synchronization algorithm and the group membership protocol. Finally, we summarize our experience in the last section and give directions for future work.

II. A BRIEF DESCRIPTION OF PVS

The PVS system combines an expressive specification language with an interactive proof checker and has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithms and architectures concerning fault-tolerance; see [12] for an overview. This section provides a brief description of the PVS language and prover, and introduces some of the concepts used in this paper; a more thorough introduction is given in [4].

The PVS specification language builds on classical typed higher-order logic with the usual base types, `bool`, `nat`, `integer`, `real`, among others, and the function type constructor `[A -> B]`. The type system of PVS is augmented with *dependent types* and *abstract data types*.

In PVS, predicates over some type `A` are, as usual, boolean-valued functions on `A`, and `pred[A]` is an abbreviation for the function type `[A -> bool]`. A distinctive feature of the PVS specification language are *predicate subtypes*: the subtype `{x:A | P(x)}` consists of exactly those elements of type `A` satisfying predicate `P`. One can use the expression `(P)` to abbreviate the subtype induced by the predicate `P`. Since sets can be described by their characteristic predicates, the expression `{x:A | P(x)}` can also be used to denote the set of elements satisfying `P`, and consequently `set[A]` is just a notational variant of `pred[A]`. Hence, `x:(P)` also means that `x` is an element of the *set* `P`.

Predicate subtypes are used, for instance, for explicitly constraining domains and ranges of operations in a specification, and to further qualify universally quantified variables in lemmas and theorems. As an example, the (uninterpreted) predicate `nonfaulty?(t)` describes whether or not a processor `p` is non-faulty at a given time *t*. The theorem `agreement` expresses the fact that at all times, all non-faulty processors agree on their membership views.

```
agreement : THEOREM
  FORALL (p,q:(nonfaulty?(t))):
    membership(t,p) = membership(t,q)
```

The theorem could also be stated without predicate subtypes using an implication with the non-faultiness constraint for `p` and `q` as the hypothesis part. However, the predicate subtype variant is preferable in many cases as it can help automating proofs that apply the agreement theorem: a crucial aspect in proof automation is to find suitable instances of variables. The annotation of type information to quantified variables enables the PVS prover to restrict the search for adequate values for `p` and `q` to those for which the predicate `nonfaulty?(t)` is known to hold.

In general, type-checking with predicate subtypes is undecidable; the type-checker generates proof obligations, so-called *type correctness conditions* (TCCs) in cases where type conflicts cannot immediately be resolved. The use of predicate subtypes frequently gives rise to TCCs: for example, if a function with a domain type `{x:A | P(x)}` is applied to an element `a` of type `A` a *subtype TCC* is generated that requires a proof of `P(a)`. A large number of TCCs are discharged by specialized proof strategies, and a PVS expression is not considered to be fully type-checked until all generated TCCs have been proved.

PVS specifications are packaged as *theories* that can be parametric in types and constants. A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts for a large number of theories. As an example, the specification below shows the theory `synchronous_system` that serves as a foundation to describe a distributed algorithm as a synchronous system.

```
synchronous_system
[(IMPORTING types)
 State     : TYPE+,
 Message   : TYPE+,
 InitState : TYPE FROM State,
 initialstate : [Proc -> InitState],
 sender    : [Slot -> Proc],
 trans     : [Proc,Message ->
                [State->State]],
 msg       : [Proc,Proc,Slot->Message]
] : THEORY
BEGIN
  slot : VAR Slot
  p    : VAR Proc
  run(slot)(p) : RECURSIVE State =
    IF slot = 0 THEN initialstate(p)
      ELSE
        LET q = sender(slot-1),
            s = run(slot - 1)
        IN
          trans(p,msg(p,q,slot-1))(s(p))
    ENDIF
      MEASURE slot
END synchronous_system
```

The theory defines a recursive function `run` that computes the state of a processor `p` in a given Slot `slot`. It is based on a state transition function `trans` which is intended to model the behaviour of a processor depending on the current state `s(p)` and the message `p` has received, as described by the function `msg`. Note that these functions are parameters of the theory. Thus the theory captures the essence of the synchronous behaviour of a systems and can be applied to model a given synchronous algorithm, such as the group membership algorithm as described in the next section, by instantiating the parameters with concrete values.

A theory can use the definitions and theorems of another theory by *importing* it. Parameterized theories can be imported in either of two ways: first, one instantiates the theory by providing actual values for the formal parameters, or, second, the theory is imported without any instantiation. In the latter case all possible instantiations of the imported theory

may be used; in the case of ambiguities, actual values can be provided to any used definition if necessary.

In the sequel, we do not always present the complete theory declarations but only the most important definitions and theorems. We do include, however, parts of the context of the definitions such as theory parameters or variable declarations as *comments*; these are marked with the symbol % at the beginning of a line.

For instance, in the following axiomatization of the maximum drift of a processor's clock, `rho` is a real-valued theory parameter, `C` is a variable ranging over the set of clocks, and t_1 and t_2 are variables of type `realtime`. To increase the readability of PVS specifications we liberally modify the syntax by replacing some ASCII codings with a more familiar mathematical notation.

```
% ρ      : {x:real | 0 ≤ x ∧ x < 1}
% C      : VAR Clock
% t1,t2  : VAR realtime
max_drift : AXIOM
  |C(t1)-C(t2) - (t1-t2)| ≤ ρ/2*|t1-t2|
```

The PVS language supports many features for making specifications more readable. For example, function and operator names can be overloaded. Moreover, variables, such as `C`, t_1 and t_2 in the axiom `max_drift` above, can occur free in formulae and are implicitly universally quantified.

Finally, we sketch some characteristics of the PVS prover. Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. PVS has an LCF-like strategy language for combining inference steps into more powerful proof strategies. The strategy `GRIND`, for example, combines rewriting with propositional simplification using BDDs and decision procedures. The most comprehensive strategies manage to generate proofs fully automatically.

III. FORMAL ANALYSIS OF TTA

The *Time-Triggered Architecture* (TTA) provides an integrated set of services for the implementation of dependable distributed real-time systems [6, 7]. It has been developed by the University of Vienna over the past twenty years and is now commercially promoted by TTTech. TTA is intended for devices controlling safety-critical electronic systems without mechanical backup, so-called "by-wire" systems such as those for automotive steering, braking, and suspension control [18]. TTA has been evaluated in two recent European projects – "Time-Triggered Architecture" (Esprit OMI program) and "X-by-Wire" (Brite EuRam program) – that have applied TTA to several prototype applications, including brake-by-wire and steer-by-wire. Recently, Audi and Honeywell have decided to utilize TTA as the basic platform for new safety-critical automotive and avionics applications, respectively; other car manufacturers are adopting the time-triggered paradigm with somewhat different architectures.

The Time-Triggered Protocol (TTP) [8] is the core of the communication level of TTA. In TTP, several distinct services, such as clock synchronization, group membership or redundancy management, are tightly integrated. For instance, there are no distinct phases for exchanging messages for clock synchronization or message acknowledgement; instead, those services are realized as side effects of ordinary, scheduled message exchanges. This tight integration makes it particularly difficult to carry out a formal analysis. It is necessary to first isolate the core algorithms that provide these services from the integrated protocol by abstracting from features that are irrelevant to the algorithm under study. Furthermore, the existing description of TTP [22] 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. We have examined two of the most important services of TTP, clock synchronization and group membership, and used the PVS verification system [12] to formally analyse their properties.

A. Ground model of TTA

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 behaviour 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, 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. 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 order to describe the state of a node at particular clock times we introduce a function $sched(r)$ 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 *sched* by recursively summing up the duration of the slots:

```
sched(r:Slot) : RECURSIVE Clocktime =
  IF r = 0 THEN system_start_time
    ELSE sched(r-1) + duration(r-1)
  ENDIF MEASURE r
```

The state of a node p at a certain clock time T is given by a function *state*. Let $comm\_dur(r)$ 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:

```
% r: VAR Slot;  T: VAR Clocktime;
% p: VAR Proc
communication_phase_def : AXIOM
  sched(r)≤T ∧ T<sched(r)+comm_dur(r) ⇒
    state(T)(p) = state(sched(r))(p)
```

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 behaviour is described by a state transition function *trans* such that $trans(p,m)(s)$ 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 $sched(r+1)$ node p has changed into a new state. By the start of the first slot, p is in its initial state. Here, the function $msg(p,T)$ models the message p has received at clock time T.

```
state_transition_def : AXIOM
 state(sched(r))(p) =
  IF r = 0 THEN initialstate(p)
    ELSE trans(p,msg(p)(T))(state(T)(p))
      WHERE T = sched(r-1)+comm_dur(r-1)
  ENDIF
```

So far we have described the general behaviour 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 captures, for instance, the algorithms for clock synchronization and group membership, which are described in the following sections.

B. Clock synchronization

Distributed dependable real-time systems crucially depend on fault-tolerant clock synchronization. This is particularly true in distributed architectures like TTA in which processors (or nodes) perform their actions according to a predetermined, static schedule. Obviously, clock synchronization is a central element of a 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.

Clock synchronization algorithms have been a matter of particular interest for formal analysis since years. Schneider [19] has observed 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 EDHM, the predecessor to PVS, to mechanically verify Schneider's proof [20, 21]. Miner [11] significantly improved Shankar's verification (by weakening its assumptions) and recast it in PVS.

For the actual verification of the clock synchronization algorithm of TTA, major emphasis has been given to making use of the cumulative development described above. 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. The specification and verification system PVS which has been used as mechanical proof assistant directly supports such an approach: parameters of theories can be constrained to satisfy certain assumptions and when using concrete instances of theorems occurring in such theories PVS serves as a book keeper that requires proofs for the concrete values to satisfy all the assumptions.

The TTA algorithm belongs to the class of averaging algorithms and is based on a variant [9] of the Lundelius-Lynch algorithm [10]. However, for optimization reasons it contains several details that complicate the direct application of existing work. For example, processors do not have access to the clock values of all other processors, but only to some of them (a possibly different set at each processor) because time difference values are stored in a queue of length four. Thus, in order to verify the TTA algorithm by showing it to be an instance of the previously verified generic derivations, those derivations had to be generalized to accommodate its peculiarities [13].

A processor's clock, more precisely its *physical clock*, 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 calculate an adjustment for the local clock. Since, in TTP, 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) [9] 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.

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 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.

```
%  s : VAR State
adj(s) : Clocktime =
  current_corr(s) + total_corr(s)
LC(p,s,t) : Clocktime =
  PC_p(t)  +  adj(s)
```

Here, *current_corr* and *total_corr* are two registers that contain the value of the most recently calculated clock adjustment and the sum of all adjustments calculated so far, respectively. These registers are modeled as parts of the internal state of a node, as well as, e. g., the stack of time difference values.

A clock synchronization protocol implements a *virtual clock* 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_agreement : THEOREM
$|VC_p(t) - VC_q(t)| \leq \delta$

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 ith synchronization interval is obtained by adding the ith adjustment to the reading of p's physical clock:

```
% i: VAR Round
```
$IC_p^i(t)$ **: Clocktime =** $PC_p(t) + adj_p^i$

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

VC_defn : AXIOM
$t_p^i \leq t \ \wedge \ t < t_p^{i+1} \ \Rightarrow \ VC_p(t) = IC_p^i(t)$

Here, t_p^i denotes the start time of the ith 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:

adj_p^i **: Clocktime =**
`IF i = 0 THEN 0 ELSE` $Cfn(p, \Theta_p^i) - PC_p(t_p^i)$
`ENDIF`

Schneider [19] 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 behaviour 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.

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.

The formal verification of Schneider's abstract properties in PVS turned out to be quite challenging, especially because TTP's special 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.

C. Group membership

Group membership is another central service of TTP as it provides to all non-faulty processors a consistent view of which nodes are operational and which are not at any given moment. Distributed fault-tolerant algorithms are inherently difficult to reason about, since careful attention has to be drawn to faults and failed components. In order to make formal verification feasible it is essential that the various aspects of an algorithm are specified and verified at appropriate levels of abstraction that capture the essence of the property under study and abstract from irrelevant details. In contrast to the clock synchronization service described above, the group membership algorithm [1] is modeled as a synchronous system – see the corresponding PVS theory in Sect. II. It abstracts from the clock synchronization service that justifies the synchrony assumption. Its verification is significantly more difficult than other fault-tolerant algorithms because information about the failure of processors is not available immediately but only with a certain delay. Therefore one has to be very careful when reasoning about possibly faulty components.

Every processor p maintains a set $\mathtt{mem}_p^t$—the membership set of processor p—that contains all processors that p considers operational at time t. In slot t the processor with label $t \bmod n$ is the broadcaster. In addition to the message data, the broadcaster sends those parts of its internal state that are critical for the protocol to work properly. More precisely, a CRC checksum that is calculated over the message data and the critical state information, which includes the membership set, is appended to the message. For the analysis of the group membership algorithm it is sufficient to assume that a message contains the broadcaster's local view $\mathtt{mem}_b^t$ on the membership.

As the order of messages is statically defined there is no need for special membership messages. Instead, a successfully received message is interpreted as a life-sign of the sender and a receiver will maintain the broadcaster in its local membership set if it agrees with the broadcaster's critical state information and hence with its membership set. Conversely, if a processor does not receive an expected message or does not agree with the broadcaster's view on the membership, the broadcaster will be considered faulty and the receiver removes it from its membership set.

The group membership algorithm is designed to operate in the presence of faults. A processor can be *send-faulty*, in which case it will fail to broadcast in its next slot, while a *receive-faulty* processor will not succeed in receiving the message of the next non-faulty processor. This restricted fault model is appropriate since other protocol services of TTP ensure that other fault modes manifest themselves as either send or receive faults by enforcing a faulty processor to fail silently. For example, the bus guardian, a special hardware element of the TTP controller, prevents a processor that has lost synchrony of its clock from accessing the broadcast bus outside its designated slots. We use $\mathcal{NF}^t$ to denote the set of non-faulty processors at time t, and $p \notin \mathcal{NF}^t$ indicates that p is either send-faulty or receive-faulty at time t.

The task of a group membership algorithm is to diagnose the failure of a faulty processor and to inform all non-faulty processors about it. In order to cause a broadcaster to realize that it is send-faulty the TTP group membership algorithm uses an (implicit) acknowledgment mechanism. A processor p that is the broadcaster in slot t checks whether the next non-faulty broadcaster, say q, that sends in the next slot has the same membership set as q and in particular contains p in its membership set. If so, p can conclude that its broadcast was successful. Otherwise, either p failed to broadcast or q is receive-faulty. To resolve this ambiguity p waits for the next non-faulty broadcaster following q, say r. If r contains p in its membership set but not q while having the same view considering other processors, the original message of p was sent correctly and q failed. If p is not in r's membership set, but q is (and the rest of the membership sets of p and r are the same), then q and r agree that p failed to send. In this case, p will remove itself from its own membership set and fail silently.

A similar mechanism could be used for diagnosing receive faults: if a processor p does not receive an expected message it could check whether the next non-faulty broadcaster maintained the original sender in its membership set in which case p must realize that it has suffered from a receive fault. However, TTP employs a slightly different mechanism that is also used to avoid the formation of disjoint cliques at the same time. A clique is a group of processors where agreement on the current state is reached only within the group. Each processor p maintains two counters, $\mathtt{acc}_p^t$ and $\mathtt{rej}_p^t$, which keep track of how many messages p has *accepted* (successfully received) and *rejected*, respectively. A processor p will increment the counter $\mathtt{rej}_p^t$ if p does not agree with the

broadcaster's view on the membership. In p's next broadcast slot it checks whether it has accepted more messages in the last round than it has rejected. If so, p resets the counters and broadcasts; the other case indicates that p suffered from a receive fault and therefore p removes itself from the membership and by not broadcasting its message p can inform the other processors about its failure.

The group membership algorithm has to fulfill the following major correctness requirements:

Validity: At all times, non-faulty processors should have all and only the non-faulty processors in their membership sets, while faulty processors should have removed themselves from their sets. This requirement is, however, impossible to satisfy as it may take some time to diagnose the faultiness of a processor. We therefore must allow a single faulty processor to be included in the membership sets of non-faulty processors, while faulty processors may have (a subset of) the non-faulty processors plus themselves in their sets.

validity : THEOREM
$$(\forall(p:(\mathcal{NF}^t)):\mathsf{mem}_p^t = \mathcal{NF}^t \ \vee \ \exists x: x \notin \mathcal{NF}^t \wedge \mathsf{mem}_p^t = \mathcal{NF}^t \cup \{x\}) \wedge \forall p: p \notin \mathcal{NF}^t \ \Rightarrow \ (p \notin \mathsf{mem}_p^t \ \vee \ \mathsf{mem}_p^t \subseteq \mathcal{NF}^t \cup \{p\})$$

Agreement: All non-faulty processors should have the same membership sets.

agreement : THEOREM
$$\forall(p,q:(\mathcal{NF}^t)):\mathsf{mem}_p^t = \mathsf{mem}_q^t$$

The requirements *validity* and *agreement* express properties that should hold for all reachable states of the system. Such invariants, or *safety-properties*, are usually verified by some form of induction proof. In order to establish the induction step, however, one generally has to strengthen the invariant because often enough the property of interest is not inductive. Usually repeated strengthening is necessary before an inductive invariant is found and although some of the strengthening can be generated automatically this becomes the main task when performing a mechanized verification. Experience with a membership algorithm similar to that of TTP [5] showed that this verification strategy is infeasible for our purpose.

Therefore, we take a different approach proposed by J. Rushby: instead of expressing the correctness property as one large conjunction, we use a set of disjunctively connected formulas that can be seen as the description of an abstract state machine [17]. Each disjunct contains the desired property and represents a particular configuration the membership algorithm can reach. To establish the correctness of the algorithm one has to show that at every point in time the system is in one of these configurations.

Thus, the main part of the proof can be represented as a *configuration diagram*. The diagram for the group membership algorithm is shown in Fig. 1. The nodes of the diagram represent the *configurations*, and arrows denote transitions from one configuration to others and are labeled with transition conditions. Configurations are parameterized by the time t and describe the global state the system is in. Configurations can have additional parameters such as processors (x, y, ...) that behave differently from the rest of system, or additional entities necessary to describe the system state. The labels of transitions express the preconditions for the system to move from one configuration to another. For example, the label $b = x$ from the transition from *latent* to *excluded* means that the system takes this transition if x is the current broadcaster, while a transition with the label *dead* $\neq x$ is taken whenever the current broadcaster is already faulty but different from x. The transition conditions leading from one configuration need not necessarily be disjoint, but one has to show that they are complete in the sense that their disjunction is true.

The diagram can be developed step-by-step. One usually starts by defining some initial configuration or the one in which the system stays under normal circumstances, i. e. as long as no fault occurs. For TTP, this central configuration is the one labeled *stable*. By symbolically evaluating the algorithm in the current configuration and by splitting on possible cases we generate some new configurations, and the transitions from the original configuration are labeled with the appropriate conditions. By repeatedly applying this construction on each transition and each new configuration one aims to develop a closed diagram. This approach can be seen as a symbolic forward exploration of the state space of the membership algorithm. To prove safety properties like *validity* or *agreement* one then has to demonstrate that every configuration implies the desired property and that the disjunction of the transition conditions leading from any one configuration evaluates to true; this ensures that there is no other configuration the system can possibly get into.

For the TTP group membership algorithm, we have formally proved both the safety properties *validity* and *agreement* mentioned above and a liveness property that states that a faulty processor will eventually remove itself from the membership. All definitions and proofs have been developed and mechanically checked with PVS [14].

D. Integration of services

As mentioned above, we have analysed the algorithms for clock synchronization and group membership separately from each other. While these analyses are valuable in their own right, it is the integrated protocol that is implemented in hardware and run in a TTP controller chip. Hence, the question arises whether the results of the isolated analyses still hold for the integration. In fact, a closer look at the formal models reveals that clock synchronization and group membership depend on each other. On the one hand, there is a hierarchical dependency in our treatment: the verification of the group membership algorithm is carried out at the level of a synchronous system, where one assumes that all processors are perfectly synchronized and run in lock-step. This assumption abstracts from a synchronization mechanism and therefore the proofs for group membership depend on the correctness of the clock synchronization service. On the other hand, the clock synchronization algorithm of TTP differs from many standard algorithms in several ways. Most

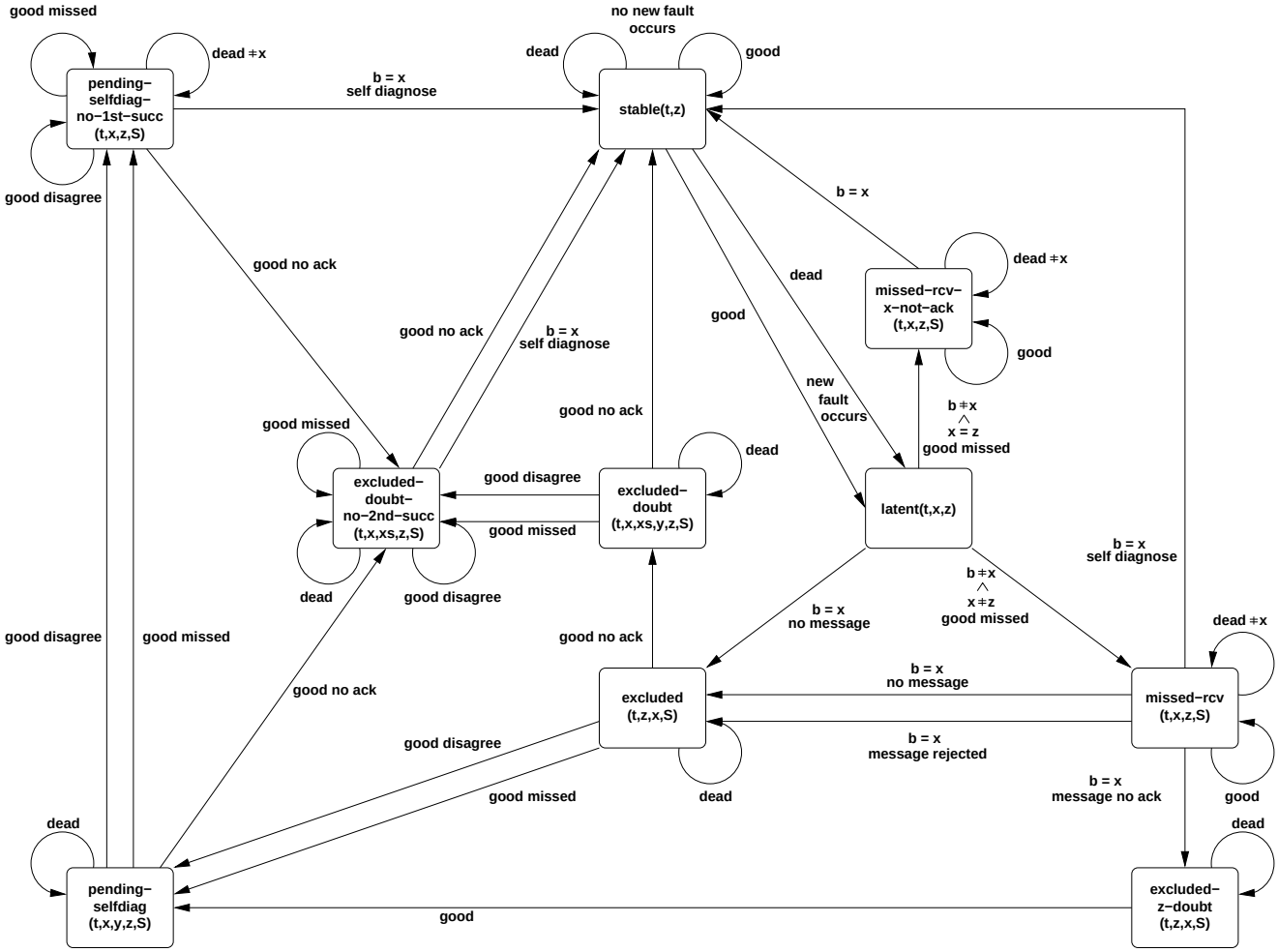

Fig. 1: Configuration diagram for the TTP group membership algorithm.

importantly, the way a processor obtains information about the clock readings of other processors is totally integrated into the exchange of data messages. Since a processor only accepts messages from processors that it considers to belong to the same membership set, clock synchronization actually depends also on group membership. These apparently circular dependencies need to be broken by reorganizing the proofs so that the results previously obtained by analysing the individual services in isolation can also be established for the integrated services. In the following, we sketch the major guidelines how this can be achieved.

The first step is to split up the analysis into a series of successive intervals, corresponding to the synchronization intervals used in the verification of the clock synchronization algorithm. The ultimate goal is then to prove, by induction on the synchronization interval *i*, that for all intervals both the clock synchronization property and the group membership property hold:

```
% i: VAR SyncRound
clocksync_and_membership : THEOREM
  clock_sync_prop(i)∧membership_prop(i)
```

The membership part of this theorem is trivially true, since the proof for the group membership property as described earlier is actually independent of synchronization intervals: the property holds for all slots and hence also for all intervals. As for the clock synchronization part, things are a bit more involved. From an abstract point of view, the proof of the induction step of the clock synchronization property proceeds according to the following lemma:

```
clocksync_step : LEMMA
  clock_sync_req(i)∧clock_sync_prop(i)
    ⇒clock_sync_prop(i+1)
```

This lemma expresses that fact that, given the clocks are synchronized during the *i*th interval and certain requirements are met in this interval, the synchronization property will hold in the interval $i+1$. The expression *clock_sync_req*(i) captures the requirements that are necessary for the synchronization algorithm to work properly. In the case of TTP, it is, for example, necessary that sufficiently many messages will be accepted by any non-faulty processor in order to produce adequate clock readings on its stack of time difference measurements. This requirement directly relies on the availability of the group membership service. Hence, we need to

show that the membership property ensures that the requirements for clock synchronization can be fulfilled:

```
membership_provides_cs_req : LEMMA
  membership_prop(i)∧clock_sync_prop(i)
    ⇒clock_sync_req(i)
```

There is an additional conjunct, *clock_sync_prop*(i), in the hypothesis of the lemma above. This is due to the fact that the membership property and the requirements for clock synchronization are expressed at different levels of abstraction. As explained earlier, group membership is dealt with at the synchronous system level, where the notion of time is abstracted away. In contrast to this, the clock synchronization requirements are expressed in terms of the ground model of TTA, cf. Sect. III.A., which explicitly deals with timing issues. It is therefore necessary to "map" the proof of group membership down to the time-triggered system level. This transformation, however, requires that synchronized clocks are present [16].

Together with the hypothesis that clocks are initially synchronized these lemmas are sufficient to prove the desired theorem of integrated synchronization and membership services.

The outlined approach deals with the dependencies between group membership and clock synchronization in two ways of abstraction. First, the dependency of membership on clock synchronization is resolved by describing the group membership algorithm at the abstract level of synchronous systems, where synchronized clocks are assumed. Second, the clock synchronization algorithm is parameterized by an abstract assumption that captures the essence of what is required from the membership service. The benefit of this approach is that there is no need to perform a double or parallel induction to accomplish the proof of the integration theorem. In fact, the two properties can be analysed more or less independently from each other and the interdependencies of the two services can be clearly isolated and resolved separately.

IV. CONCLUSIONS

In this paper we have discussed how formal analysis may assist in ascertaining safety-critical dependability properties of the Time-Triggered Architecture.

Besides establishing, by giving a mathematical proof, the mere fact that some important properties do hold for a system (more precisely: the model of the system), a formal analysis can have further benefits. Indeed, some consider proving a system or a program correct rather uninteresting. What is more valuable is that a careful formal analysis can reveal imprecise requirements, inconsistencies, or even bugs. Our experience with formally analysing several services of TTP shows that also the first of the two steps mentioned at the beginning, the modeling phase, is beneficial to the system itself. The need of precise statements within the formal model can yield to a substantially improved description of the algorithms by clarifying issues initially missing or left vague. Furthermore, developing a formal model can contribute to the understanding of the service under study. The group membership algorithm of TTA, for example, is extremely complex, because it also incorporates other services, such as implicit acknowledgement of messages. The formal analysis of the algorithm yielded considerable insight into its operation, and the model has also proven useful to explain how it works (or in which cases it does not).

The work discussed here is still ongoing; we expect to expand it by investigating further system properties and to adapt it to a variant of the architecture in the context of a forthcoming new EU project.

V. ACKNOWLEDGMENTS

This work was partly supported by the European Commission under project ESPRIT OMI 23396 "Time-Triggered Architecture (TTA)".

VI. REFERENCES

[1] G. Bauer and M. Paulitsch. An Investigation of Membership and Clique Avoidance in TTP/C. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems*. IEEE, Oct. 2000. To appear.

[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] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, London, 1999.

[4] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[5] S. Katz, P. Lincoln, and J. Rushby. Low-Overhead Time-Triggered Group Membership. In M. Mavronicolas and P. Tsigas, editors, *11th International Workshop on Distributed Algorithms: WDAG'97*, volume 1320 of *Lecture Notes in Computer Science*, pages 155–169, 1997.

[6] 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-Verlag, 1995.

[7] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Engineering and Computer Science. Kluwer, 1997.

[8] H. Kopetz and G. Grünsteidl. TTP – A Time Triggered Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, 1994.

[9] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. Computers*, 36(8):933–940, 1987.

[10] J. Lundelius-Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, Apr. 1988.

[11] P. S. Miner. Verification of Fault-Tolerant Clock Synchronization Systems. NASA Technical Paper 3349, NASA Langley Research Center, January 1994.

[12] 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.

[13] H. Pfeifer, D. Schwier, and F. W. von Henke. Formal Verification for Time-Triggered Clock Synchronization. In C. B. Weinstock and J. Rushby, editors, *Dependable Computing for Critical Applications 7*, volume 12 of *Dependable Computing and Fault-Tolerant Systems*, pages 207–226. IEEE Computer Society, January 1999.

[14] Holger Pfeifer. Formal Verification of the TTP Group Membership Algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Methods for Distributed System Development – Proceedings of FORTE XIII / PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000. Kluwer Academic Publishers.

[15] J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. In R. Shaw, editor, *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*. Springer-Verlag, 1995.

[16] 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.

[17] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In *Computer Aided Verification (CAV 2000)*, Chicago, IL, July 2000. To appear.

[18] C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture. In J.-Y. Roger, B. Stanford-Smith, and P. T. Kidd, editors, *Advances in Information Technologies: The Business Challenge. Proceedings of EMMSEC'97*. IOS Press, 1997.

[19] F. B. Schneider. Understanding Protocols for Byzantine Clock Synchronization. Technical Report 87-859, Cornell University, Aug. 1987.

[20] N. Shankar. Mechanical Verification of a Schematic Byzantine Clock Synchronization Algorithm. Technical Report CR-4386, NASA, 1991.

[21] N. Shankar. Mechanical Verification of a Generalized Protocol for Byzantine Fault-Tolerant Clock Synchronization. In J. Vytopil, 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.

[22] TTTech. Specification of the TTP/C Protocol. Available on request from TTTech, `http://www.tttech.com/`, 2000.