Updated version with correction of minor typographic errors of a paper published in:
Proc. of FORTE XIII / PSTV XX, pp. 3–18, Pisa, Italy, Oct. 2000.
© 2000 Kluwer Academic Publishers

FORMAL VERIFICATION OF THE TTP GROUP MEMBERSHIP ALGORITHM

Holger Pfeifer
Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
pfeifer@informatik.uni-ulm.de

Abstract This paper describes the formal verification of a fault-tolerant group membership algorithm that constitutes one of the central services of the Time-Triggered Protocol (TTP). The group membership algorithm is formally specified and verified using a diagrammatic representation of the algorithm. We describe the stepwise development of the diagram and outline the main part of the correctness proof. The verification has been mechanically checked with the PVS theorem prover.

Keywords: Deductive verification, Time-Triggered Architecture, fault-tolerant distributed algorithms, safety-critical control.

1. INTRODUCTION

The Time-Triggered Architecture (TTA) developed by the University of Vienna and TTTech provides an integrated set of services for dependable distributed real-time systems [5, 6]. 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 [15]. 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 [3, 12].

The Time-Triggered Protocol (TTP) [7] is the core of the communication level of TTA. Group membership is 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. 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. The group membership algo-

rithm [1] is modeled as a synchronous system and 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 failed components. Verification of safety properties, like the requirement that all (non-faulty) processors of a system should have the same opinion about the current membership status of other processors, is typically accomplished by an 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 strengthenings can be generated automatically this becomes the main task when performing a mechanized verification. Experience with a membership algorithm similar to that of TTP [4] showed that this verification strategy is infeasible for our purpose.

Therefore, we take a different approach recently proposed by John 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 [14]. 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. For the TTP group membership algorithm, we have formally proved both an agreement property, i.e., that every non-faulty node considers the same set of processors to be part of the membership, 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 the PVS specification and verification system [10]. We present fragments of the PVS specification in this paper. To increase readability, however, the syntax has liberally been modified by replacing some ASCII codings with a more familiar mathematical notation. The full specification together with all proofs can be obtained via WWW [16].

The next section explains the group membership algorithm of TTP and provides a formal specification. Section 3 illustrates the approach we take to formally verify the algorithm, and Section 4 describes the main parts of the proof. The last section contains concluding remarks and directions for future work.

2. GROUP MEMBERSHIP IN TTP

In TTP, access to the broadcast bus is implemented by a *time division multiple access (TDMA)* schema: each processor is assigned a certain time interval, a so-called *slot*, in which it is allowed to send a message on the bus while the other processors listen. Slots are numbered and can be seen as an abstraction

of a global time base that is provided by a separate clock synchronization algorithm [8, 11]. In our model we assume a set $\mathtt{proc}$ of n processors, labeled $0, 1, \ldots, n-1$, that are arranged in a logical ring. The period of n successive slots is called a *TDMA round*.

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, denoted $\mathit{broadcaster}(t)$. 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.[1] 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. Furthermore, sends_b^t describes that the current broadcaster b sends a message on the bus, while $\mathit{arrives}_p^t$ means that the message sent by the broadcaster arrives at the receiver p.

A non-faulty broadcaster b will only send a message if b is contained in its own membership set; if b has removed itself from the membership set (due to diagnosing a fault) it will stop sending message in its broadcast slot. The

[1] Technically, the receiver calculates a CRC checksum over the received message data and its own critical state information and compares the result with the checksum sent by the broadcaster.

following specification [1] shows the axiomatization of $sends_b^t$ as defined in PVS:[2]

```
[1]
NF^t       : set[proc]
sends_b^t  : bool
arrives_p^t : bool

sending : AXIOM
  LET b = broadcaster(t) IN
    b ∈ NF^t  ∧  b ∈ mem_b^t  ⇒  sends_b^t

fail_silence : AXIOM
  LET b = broadcaster(t) IN
    b ∉ mem_b^t  ⇒  ¬sends_b^t
```

A message sent by the current broadcaster b will arrive at a non-faulty processor p . Of course, there is no generation of spontaneous messages and hence messages arrive only if they have been sent. These axioms also imply that broadcasts are consistent: a message arrives either at all non-faulty processors or, if the broadcaster is send-faulty, at none of them. The PVS specification is given in [2].

```
[2]
arrival : AXIOM
  LET b = broadcaster(t) IN
    sends_b^t  ∧  p ∈ NF^t  ⇒  arrives_p^t

nonarrival : AXIOM
  LET b = broadcaster(t) IN
    ¬sends_b^t  ⇒  ¬arrives_p^t
```

In our model we consider faults to occur only if they affect the system in the next slot; e. g. a send fault of a processor p is not considered to occur until immediately before p's broadcasting slot. If the broadcaster at time $t+1$ becomes faulty in its sending slot it will fail to successfully send a message. Similarly, a newly faulty processor will fail to receive a message. Note that the behaviour of faulty processors is unspecified for the slots after the fault occurred: processors may or may not succeed in sending or receiving subsequent messages.

```
[3]
faulty_broadcaster : AXIOM
  LET b = broadcaster(t+1) IN
    b ∈ NF^t  ∧  b ∉ NF^{t+1}  ⇒  ¬sends_b^{t+1}

faulty_receiver : AXIOM
  LET b = broadcaster(t+1) IN
    p ∈ NF^t  ∧  p ∉ NF^{t+1}  ∧  p ≠ b  ⇒  b ∈ NF^{t+1}  ∧  ¬arrives_p^{t+1}
```

[2]PVS allows variables such as t to occur free in formulas; these are implicitly universally quantified. Moreover, the type of variables can be dropped if it has been introduced to PVS by a corresponding variable declaration. Throughout this paper, we use b, p, q, x, y, and z to denote processors (of type `proc`), and t and s are of type `time`.

Finally, once a processor becomes faulty it will be considered faulty forever.

```
faults_latch : AXIOM                                                    4
  p ∉ NF^t  ⇒  p ∉ NF^{t+1}
```

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[3] 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, $\texttt{acc}_p^t$ and $\texttt{rej}_p^t$, which keep track of how many messages p has *accepted* (successfully received) and *rejected*, respectively. A processor p will increment the counter $\texttt{rej}_p^t$ 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.

Formally, the group membership algorithm is described by a set of guarded commands. In every slot t, every processor executes exactly one of these commands. The guards are evaluated in a top-down order. The formal description involves two additional boolean state variables, $\texttt{prev}_p^t$ and

[3] More generally, if there are already faulty processors that were scheduled to send between p and q, the latter is the broadcaster in slot $t+i$ for some $i > 1$.

$\mathtt{doubt}^t_p$. If a processor p was the previous broadcaster and now waits for being acknowledged, $\mathtt{prev}_p$ is set to true, while $\mathtt{doubt}_p$ is true if p did not get acknowledged by its successor and waits for the second successor to resolve the conflict. In this case, the variable $\mathtt{succ}^t_p$ holds p's first successor which refused to acknowledge p. In the following definition state components that are not mentioned explicitly do not change.

Broadcaster: Let b be the current broadcaster, i. e. $b = t \bmod n$.

(1)	$\mathtt{acc}^t_b > \mathtt{rej}^t_b$	$\rightarrow$	$\mathtt{mem}^{t+1}_b := \mathtt{mem}^t_b$, $\mathtt{prev}^{t+1}_p := \mathsf{T}$, $\mathtt{acc}^{t+1}_b := 1$, $\mathtt{rej}^{t+1}_b := 0$
(2)	$\mathtt{acc}^t_b \leq \mathtt{rej}^t_b$	$\rightarrow$	$\mathtt{mem}^{t+1}_b := \mathtt{mem}^t_b \setminus \{b\}$

Receiver: Every processor p different from the current broadcaster executes the first of the following commands whose guard evaluates to true:

(3)	$p \notin \mathtt{mem}^t_p$	$\rightarrow$	no change
(4)	$\mathtt{prev}^t_p \wedge \mathit{arrives}^t_p \wedge$ $\mathtt{mem}^t_b = \mathtt{mem}^t_p \cup \{p\}$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p$, $\mathtt{prev}^{t+1}_p := \mathsf{F}$, $\mathtt{acc}^{t+1}_p := \mathtt{acc}^t_p + 1$
(5)	$\mathtt{prev}^t_p \wedge \mathit{arrives}^t_p \wedge$ $\mathtt{mem}^t_b = \mathtt{mem}^t_p \setminus \{p\}$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{prev}^{t+1}_p := \mathsf{F}$, $\mathtt{doubt}^{t+1}_p := \mathsf{T}$, $\mathtt{rej}^{t+1}_p := \mathtt{rej}^t_p + 1$, $\mathtt{succ}^{t+1}_p := b$
(6)	$\mathtt{prev}^t_p \wedge \mathit{sends}^t_b$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{prev}^{t+1}_p := \mathsf{T}$, $\mathtt{rej}^{t+1}_p := \mathtt{rej}^t_p + 1$
(7)	$\mathtt{prev}^t_p$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{prev}^{t+1}_p := \mathsf{T}$
(8)	$\mathtt{doubt}^t_p \wedge \mathit{arrives}^t_p \wedge$ $\mathtt{mem}^t_b = \mathtt{mem}^t_p \cup \{p\} \setminus \{\mathtt{succ}^t_p\}$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p$, $\mathtt{doubt}^{t+1}_p := \mathsf{F}$, $\mathtt{acc}^{t+1}_p := \mathtt{acc}^t_p + 1$
(9)	$\mathtt{doubt}^t_p \wedge \mathit{arrives}^t_p \wedge$ $\mathtt{mem}^t_b = \mathtt{mem}^t_p \cup \{\mathtt{succ}^t_p\} \setminus \{p\}$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{p\}$, $\mathtt{doubt}^{t+1}_p := \mathsf{F}$, $\mathtt{acc}^{t+1}_p := \mathtt{acc}^t_p + 1$
(10)	$\mathtt{doubt}^t_p \wedge \mathit{sends}^t_b$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{doubt}^{t+1}_p := \mathsf{T}$, $\mathtt{rej}^{t+1}_p := \mathtt{rej}^t_p + 1$
(11)	$\mathtt{doubt}^t_p$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{doubt}^{t+1}_p := \mathsf{T}$
(12)	$\mathit{arrives}^t_p \wedge (\mathtt{mem}^t_p = \mathtt{mem}^t_q)$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p$, $\mathtt{acc}^{t+1}_p := \mathtt{acc}^t_p + 1$
(13)	$\neg \mathit{sends}^t_b$	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$
(14)	otherwise	$\rightarrow$	$\mathtt{mem}^{t+1}_p := \mathtt{mem}^t_p \setminus \{b\}$, $\mathtt{rej}^{t+1}_p := \mathtt{rej}^t_p + 1$

The group membership algorithm has to fulfill three 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                                                              5
```
$$p \in \mathcal{NF}^t \Rightarrow \mathrm{mem}_p^t = \mathcal{NF}^t \vee \exists x:\ x \notin \mathcal{NF}^t \wedge \mathrm{mem}_p^t = \mathcal{NF}^t \cup \{x\}$$
$$\wedge\ p \notin \mathcal{NF}^t \Rightarrow p \notin \mathrm{mem}_p^t \vee \mathrm{mem}_p^t \subseteq \mathcal{NF}^t \cup \{p\}$$

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

```
agreement : THEOREM                                                             6
```
$$p \in \mathcal{NF}^t \wedge q \in \mathcal{NF}^t \Rightarrow \mathrm{mem}_p^t = \mathrm{mem}_q^t$$

- **Self-diagnosis:** A faulty processor should eventually diagnose its fault and remove itself from its own membership set.

```
self_diagnosis : THEOREM                                                        7
```
$$x \in \mathcal{NF}^t \wedge x \notin \mathcal{NF}^{t+1} \Rightarrow \exists s:\ 0<s \wedge s \leq 2n+1 \wedge x \notin \mathrm{mem}_x^{t+s}$$

These properties are subject to two additional assumptions that constitute the *fault hypothesis*. First, as processors will be able to diagnose a fault only if no new fault occurs during that process, the specification of the Time-Triggered Protocol requires the membership algorithm to work properly only if two successive failures occur at least two TDMA rounds apart [1]. More frequent fault arrivals are dealt with by other protocol mechanisms of TTP. In our formal verification the fault arrival assumption is expressed in a different way that is adapted to the verification approach we take. We assume that a fault only occurs if the system is in a certain configuration called *stable*. If a fault occurs the system leaves this configuration and the fault arrival assumption states that no new fault occurs until the system reaches the *stable* configuration again.

```
fault_may_occur_when_stable : AXIOM                                             8
```
$$\mathrm{stable}(t,y,z) \Rightarrow \mathcal{NF}^{t+1} = \mathcal{NF}^t \vee \exists x:\ x \in \mathcal{NF}^t \wedge \mathcal{NF}^{t+1} = \mathcal{NF}^t \setminus \{x\}$$

```
no_faults_when_not_stable : AXIOM
```
$$\neg\mathrm{stable}(t,y,z) \Rightarrow \mathcal{NF}^{t+1} = \mathcal{NF}^t$$

It is necessary to rule out the trivial solution to the *validity* and *agreement* requirements where the system never returns to *stable*. Otherwise only one

fault could occur which would not even be tolerated. We are therefore forced to prove a *liveness* property saying that after a fault occurred the *stable* configuration will eventually be reached again. More specifically, we demonstrate that this is to happen no later than $2n + 1$ steps after the fault occurred.

```
liveness : THEOREM                                                          9
  stable(t,y,z)  ⇒  ∃s,y,z: 0<s ∧ s ≤ 2n+1 ∧ stable(t+s,y,z)
```

We believe that a stronger property could also be proved, where s is constrained to be less or equal $2n$. Then the system would return to *stable* no later than two TDMA rounds after a fault occurred which would yield the original fault arrival assumption of TTP. Moreover, the *self-diagnosis* requirement listed above is actually a simple corollary of *liveness*.

The second part of the fault hypothesis refers to the number of processors that are required for the algorithm to work correctly. A set of three processors would be sufficient to tolerate a single fault, however, in order to prevent a non-faulty processor from being fooled by a faulty broadcaster at least 4 processors are required to be present in the system with at least three being non-faulty at all times. Otherwise a non-faulty processor might mistakenly diagnose a fault of its own and remove itself from its membership set (thereby violating both the validity and the agreement property).

```
n    : { m:ℕ | m≥4 }              % -- number of processors             10
proc : TYPE+ = { m:ℕ | 0≤m<n }    % -- the set of processors (labels)

three_non_faulty_exist : AXIOM
  ∃x,y,z: x≠y ∧ x≠z ∧ y≠z ∧ x∈NF^t ∧ y∈NF^t ∧ z∈NF^t
```

3. APPROACH TO VERIFYING THE ALGORITHM

For the verification of the TTP group membership algorithm we apply a method recently proposed by John Rushby [14]. 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; one demonstrates that the property holds in the initial state(s) and that all state transitions preserve the property. The problem is, however, that the property of concern is not inductive in general and hence, in order to establish the proof of the induction step, one has to strengthen it by conjoining additional properties to it which themselves have to be invariants. Usually, this process has to be repeated several times before the induction proof can be accomplished. Rushby reports in [14] that the number and complexity of additional invariants that had to be discovered during the proof defeated several determined attempts to mechanically verify a group membership algorithm [4] similar to that of TTP. Because of its peculiarity of managing counters for accepted and rejected messages and the way how acknowledgment of messages

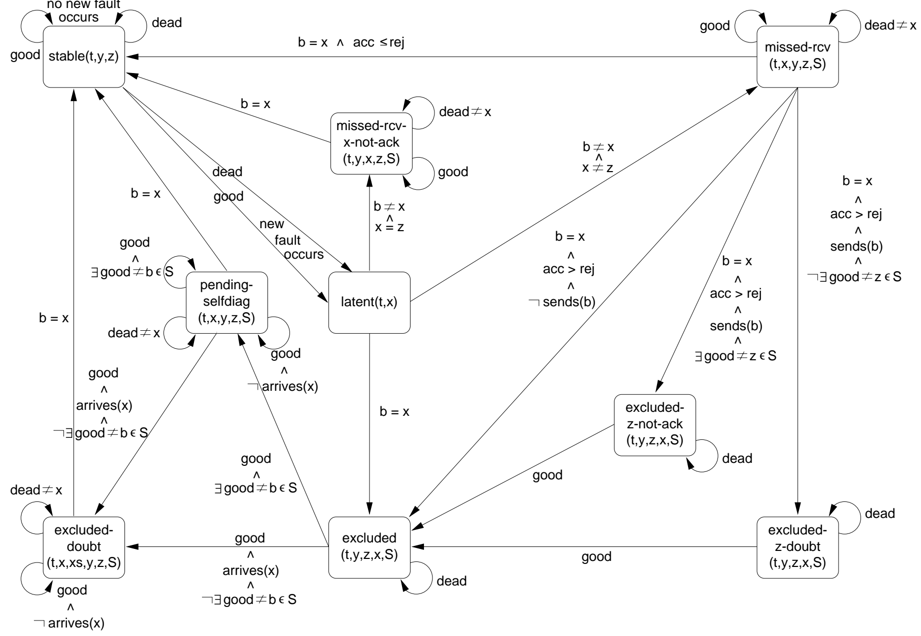

Figure 1 Configuration diagram for the TTP group membership algorithm.

is done, the group membership algorithm of TTP is considered quite tricky and a mechanical proof using the traditional approach is very likely to be even more complex.

Instead of expressing the property to be verified as a conjunction of predicates Rushby's method exploits a disjunctively connected set of formulas—which in turn may be conjunctions of predicates—to prove the property of concern. Each disjunct can be seen as the description of a certain *configuration*, a property of the current global state of the whole system. The configurations are defined such that every single configuration implies the desired property, and to verify that property one has to show that at all times 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 Figure 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. 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. In order to prove liveness properties like *self-diagnosis* one has to establish that the system can not loop forever on a configuration other than *stable*.

There are several benefits to this approach: first, the diagram can be developed incrementally and in a totally systematic way by symbolically executing one step of the algorithm in every configuration. Second, the completed diagram is a suitable means of analysing the difficult special cases of the algorithm and to explain how and why it works (or doesn't). Last, it seems that the creative steps in developing the proof can be accomplished easier than by using the traditional way of repeated invariant strengthening. The configurations as presented here generally are not invariants and are therefore identified more easily.

The next section describes how the configuration diagram for the TTP group membership algorithm is gradually developed and outlines the verification of the three correctness requirements for the TTP group membership algorithm.

4. DEVELOPING THE CONFIGURATION DIAGRAM

The system is said to be in a *stable* configuration if the membership set of all non-faulty processors p is equal to $\mathcal{NF}^t$, the set of all non-faulty processors at time t, and all faulty processors have already diagnosed their fault and thus removed themselves from their own membership set. For *stable* the two safety properties *validity* and *agreement* follow immediately from these definitions. Moreover, *stable* is the initial configuration of the system.

```
stable(t,y,z) : bool =                                                    11
      recent(t,y,z)
  ∧ ∀p:    p ∉ NF^t  ⇒  p ∉ mem_p^t
        ∧  p ∈ NF^t  ⇒     mem_p^t = NF^t
                         ∧  p=z  ⇔  acc_p^t = rej_p^t + 1
                         ∧  p≠z  ⇒  acc_p^t > rej_p^t + 1
                         ∧  prev_p^t = ⊤  ⇔  p=z
                         ∧  doubt_p^t = ⊥
initial : AXIOM
  ∃y,z: stable(0,y,z)
```

In the configuration $stable(t, y, z)$ the counters of non-faulty processors are set such that $\texttt{acc}_p^t > \texttt{rej}_p^t + 1$. This is to allow for a non-faulty processor p to cope with a send fault of one of the other broadcaster in the next round, in which case the counter $\texttt{rej}_p^t$ will be increased; this should not lead to p removing itself from its own membership set in its next sending slot, for which $\texttt{acc}_p^t > \texttt{rej}_p^t$ must hold. However, the most recent non-faulty broadcaster, say z, cannot satisfy this condition as in its sending slot z sets the counters to $\texttt{acc}_z^t = 1$ and $\texttt{rej}_z^t = 0$. Suppose z was the broadcaster at time $t - 1$ and the broadcaster at time t committed a send fault. Now z increases $\texttt{rej}_z^t$ and has $\texttt{acc}_z^t = \texttt{rej}_z^t$. If z was scheduled to be the next broadcaster at time $t + 1$ the condition $\texttt{acc}_p^t > \texttt{rej}_p^t$ would not hold and thus z would execute command (2), thereby wrongly removing itself from its membership set. In order to avoid this case, it is crucial that z cannot be the next broadcaster. Therefore we must ensure that there is another non-faulty processor, say y, that will send before z does. For similar, yet more involved reasons there has to be another non-faulty processor broadcasting before both z and y. As a result, this requires at least 4 processors to be present in the system in order for it to correctly tolerate a single fault. Finally, in *stable*, z is the only processor that has its `prev` flag set, while the `doubt` flag is cleared for all processors.

The characterization of the two most recent non-faulty broadcasters is covered by the following predicate, which also expresses that neither y nor z can be the broadcaster at time t.

```
recent(t,y,z) : bool =                                                    12
      y ∈ NF^t  ∧  z ∈ NF^t
  ∧ before(t,y,z)
  ∧ ∃p: p≠y  ∧  p ∈ NF^t  ∧  before(t,p,z)
  ∧ ∀p: before(t,zp)  ⇒  p ∉ NF^t
  ∧ ∃p: p ∈ NF^t  ∧  before(t,p,y)
  ∧ ∀p: before(t,y,p)  ∧  before(t,p,z)  ⇒  p ∉ NF^t
```

The expression $before(t, y, z)$ denotes that at time t, processor y will broadcast again before z does.

```
before(t,y,z) : bool =                                                    13
  ∀(i:time): z=broadcaster(t+i)  ⇒  ∃(s:time): s<i  ∧  y=broadcaster(t+s)
```

In order to determine the transition conditions from the *stable* configuration, we consider whether or not a new fault occurs. If no processor becomes faulty, that is $\mathcal{NF}^{t+1} = \mathcal{NF}^t$ holds, the system remains in *stable*, because neither the broadcaster nor the receivers change their membership set: the broadcaster will execute command (1), while the receivers will execute command (12), except for z which will execute command (4). However, depending on whether the current broadcaster b is non-faulty or an already faulty one, the values of the parameters of *stable* might change. In the first case, z and b now become the two most recent non-faulty broadcaster and the new configuration is $stable(t+1, z, b)$, while in the latter case nothing changes, i. e. the new configuration is $stable(t+1, y, z)$.

14

```
stable_to_stable_non_faulty : LEMMA
  LET b = broadcaster(t) IN
    stable(t,y,z)  ∧  b ∈ NF^t  ∧  NF^{t+1} = NF^t  ⇒  stable(t+1,z,b)

stable_to_stable_faulty : LEMMA
  LET b = broadcaster(t) IN
    stable(t,y,z)  ∧  b ∉ NF^t  NF^{t+1} = NF^t  ⇒  stable(t+1,y,z)
```

If a processor, say x, which was non-faulty at time t becomes faulty at time $t+1$, the same commands as above will be executed, but the system will change into a new configuration, that we call *latent*.

15

```
latent(t,x,y,z) : bool =
       x ∈ NF^{t-1}  ∧  NF^t = NF^{t-1} ∖ {x}  ∧  recent'(t,x,y,z)
  ∧  ∀p:     (p ∉ NF^t  ∧  p ≠ x)  ⇒  p ∉ mem_p^t
         ∧  (p ∈ NF^t  ∨  p = x)  ⇒     mem_p^t = NF^t ∪ {x}
                                      ∧  p = z  ⇔  acc_p^t = rej_p^t + 1
                                      ∧  p ≠ z  ⇒  acc_p^t > rej_p^t + 1
                                      ∧  prev_p^t = ⊤  ⇔  p = z
                                      ∧  doubt_p^t = ⊥
```

This configuration is very similar to *stable* with the exception that both the non-faulty processors and x now do not only contain non-faulty processors in their membership sets, but also the newly faulty processor x. It is the task of the group membership algorithm to ensure that eventually all processors, including x, become aware of x's faultiness and remove it from their membership sets and thus return to the *stable* configuration. The fault hypothesis states that no new fault will occur during that time until the system will be *stable* again. The predicate $recent'(t, x, y, z)$ differs from $recent(t, y, z)$ in that we must allow y or z be actually identical to x and hence the first two conjuncts are changed to $y \neq x \Rightarrow y \in \mathcal{NF}^t$ and $z \neq x \Rightarrow z \in \mathcal{NF}^t$, respectively.

Again, it is a simple matter to establish the properties *validity* and *agreement* for the configuration *latent*. All non-faulty processors have the same membership sets which contain only one faulty processor, and the faulty processors have already removed themselves from their membership sets, apart from x,

which still has itself in its membership. An additional verification condition, which is proved equally easily, is that *latent* is different from *stable* (and therefore faults are assumed not to occur).

As for *stable* there are two transition conditions for the system to transit to *latent*: either the broadcaster b at time $t+1$—when x becomes faulty—is an already faulty processor, or it is non-faulty. In the first case, the new configuration is *latent*$(t+1, x, z, b)$, while in the latter case the system will be in *latent*$(t+1, x, y, z)$.

```
                                                                          16
stable_to_latent_non_faulty : LEMMA
  LET b = broadcaster(t) IN
    stable(t,y,z)  ∧  x ∈ NF^t  ∧  b ∈ NF^t  ∧  NF^{t+1} = NF^t \ {x}
      ⇒ latent(t+1,x,z,b)

stable_to_latent_faulty : LEMMA
  LET b = broadcaster(t) IN
    stable(t,y,z)  ∧  x ∈ NF^t  ∧  b ∉ NF^t  ∧  NF^{t+1} = NF^t \ {x}
      ⇒ latent(t+1,x,y,z)
```

From *latent* two cases must be distinguished: one where the faulty processor x is the next broadcaster, and one where it is not. In the former case, x executes command (1), however, it will not be able to successfully transmit its message and hence the receivers will execute their command (13), thereby removing x from their membership sets. This leads to the configuration *excluded*. In the latter case, x is a (faulty) receiver and hence will not succeed to receive the message sent by the broadcaster. The other receivers will execute command (12), or (4) in the case of the previous broadcaster. On the other hand, x executes either command (6) or (14), depending on whether or not x was the previous broadcaster (z, that is) and the system moves into the configurations *missed-rcv* or *missed-rcv-x-not-ack*, respectively.

By systematically analyzing the possible cases for a given configuration one proceeds to develop the configuration diagram. Every transition either leads to a new configuration or to an already existing one. In some cases it may be necessary to generalize an existing configuration in order to establish the proof of a transition. The ultimate goal in this process is to end up with a configuration diagram which is closed. Due to lack of space we have to omit the detailed description of the remaining states and transitions of the diagram in this paper. The complete specification of the algorithm and all proofs are available via WWW [16].

Once the configuration diagram is closed the most difficult and complex part of the proof of the two safety properties *validity* and *agreement* is done. To formally established these properties one has to show basically two things: first, every configuration implies the property under consideration. This is quite easy to see as all the non-faulty processors always have the same membership sets and there is always at most one faulty processor, namely x, that has not yet

removed itself from its membership set. Second, there is no other configuration the system could possibly turn into. In other words, the transition conditions from any configuration must cover all the possible cases. Once this part of the proof is done, which is mainly a task of simple case analyses, these two facts can be combined to finally yield the desired safety properties.

The third correctness property, *self-diagnosis*, is a liveness property. Intuitively, one has to show that once the system has left *stable* it can not be trapped in one of the loops of the other configurations. Every of these configurations with a loop can be left in either of two cases: first, if the next broadcaster is non-faulty or, second, if it is x's slot to broadcast. The latter case is easy to demonstrate as every n slots a given processor will be the current broadcaster. The former case can be established using the fault model, see PVS box 10, that states that there exist at least three non-faulty processors at all times. Hence, eventually one of these will become the next broadcaster.

5. CONCLUSIONS

The group membership protocol presented here is one of a whole suite of algorithms for safety-critical real-time control implemented in the Time-Triggered Protocol. Industrial needs to minimize cost enforces the group membership algorithm to be heavily intertwined with other protocol services of TTP. We have isolated the core of the group membership algorithm and subjected it to thorough formal analysis. The proofs of the main correctness properties of the algorithm have been developed and mechanically checked with the assistance of the PVS specification and verification system. The complete verification comprises more than 160 proved lemmas and theorems and it takes almost one and a half hours to re-run all proofs even on a fast Sparc Ultra-II. These numbers might give an impression of the immense complexity of fault-tolerant group membership algorithms in general and of the TTP instance in particular. One source of this complexity is the requirement of fault-tolerance itself as one has to be very careful when reasoning about possibly failed components. Another reason why the TTP algorithm is significantly more difficult than other similar group membership protocols is that for optimization reasons there are still other services present as parts of the TTP group membership algorithm; for example, the way how self-diagnosis is accomplished in TTP by using the two special state variables $\texttt{acc}_p$ and $\texttt{rej}_p$ is in fact used for doing clique avoidance at the same time. This results in a self-stabilizing group membership algorithm, a property not considered in this paper.

The verification of the TTP group membership protocol applies a method recently proposed by John Rushby [14] who has used it to mechanically verify a similar algorithm [13]. Several attempts to formally verify that algorithm, which was flawed in its original publication [4], failed because of the number

and complexity of additional invariants needed to establish the proof. This proof method, which is quite closely related to the verification diagrams of Manna and Pnueli [2,9], made it both possible and easy to verify the (corrected) algorithm. This promising experience with an apparently difficult problem made us decide to apply Rushby's method rather than to attempt a traditional invariance proof of the correctness of the TTP algorithm, which is considered even more complicated. It turned out that one of its main advantages over the usual method of repeated invariant strengthening is that the proof can be developed incrementally and in a very systematic way. Thus, one can break down the overall proof into small manageable steps.

Further research is concerned with formally specifying other protocol services of the Time-Triggered Architecture, such as initialization or re-integration of nodes, and analyzing their interrelationships. For example, the group membership algorithm presented here is specified at the level of a synchronous system; for this model to be adequate, we have to assume fault-tolerant clock synchronization [11]. Conversely, the clock synchronization mechanism of TTP also relies on the group membership service being able to avoid the formation of cliques of processors. Finding ways to clearly identify the relationships and interfaces of the various protocol services in order to avoid these circular dependencies remain a challenging and interesting problem.

Acknowledgments

I would like to thank John Rushby for introducing me to his method and for sharing both experience and PVS theories, which served as a starting point for the verification described here. I am also grateful for the constructive comments of the anonymous referees and of my colleagues at Ulm that substantially improved the presentation. Last, but not least, Michael Paulitsch and Günter Bauer from Vienna were of great assistance in understanding all the subtle details of TTP group membership.

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] I. A. Browne, Z. Manna, and H. Sipma. Generalized Temporal Verification Diagrams. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[3] 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.

[4] S. Katz, P. Lincoln, and J. Rushby. Low-Overhead Time-Triggered Group Membership. In M. Mavronicolas and P. Tsigas, editors, *Proc. of WDAG'97*, volume 1320 of *LNCS*, pages 155–169, 1997.

[5] 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.

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

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

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

[9] Z. Manna and A. Pnueli. Temporal Verification Diagrams. In M. Hagiya and J. C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software: TACS'94*, volume 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[10] 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.

[11] 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.

[12] 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.

[13] J. Rushby. Formal Verification of a Low-Overhead Group Membership Algorithm, 2000. In Preparation.

[14] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

[15] 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.

[16] `http://www.informatik.uni-ulm.de/ki/PVS/membership.html`.