Verification of Distributed Real-Time and Fault-Tolerant Protocols

Jozef Hooman

Department of Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl
http://www.win.tue.nl/win/cs/tt/hooman/JH.html

Abstract. An assertional method to verify distributed real-time and fault-tolerant protocols is presented. To obtain mechanical support, the verification system PVS is used. General PVS theories are developed to deal with timing and failures. As a characteristic example, we verify a processor-group membership protocol, dealing with a dynamically changing network of processors and reasoning in terms of local clocks. Further we show some basic theories for the verification of the underlying synchronous atomic broadcast service.

1 Introduction

Safety-critical applications are often implemented using a hierarchy of protocols that provide a certain real-time and fault-tolerant service. For instance, starting from a network of processors with local clocks and a simple communication mechanism, usually some protocol is applied to achieve internally synchronized clocks. Based on synchronized clocks, a more powerful communication mechanism, such as multicast or atomic broadcast, can be implemented. This can then be used for higher-level services. Such a protocol stack usually forms the basis for numerous critical applications, and hence the correctness of the protocols is very important. On the other hand, the combination of distribution, local clocks, real-time, and fault-tolerance makes it extremely difficult to guarantee that these protocols indeed lead to the required service.

One method to increase the confidence in the correctness is the application of formal methods to verify (part of) the protocols. A large number of methods have been devised to specify and verify distributed real-time systems (see, e.g., [JP96]), but it is less clear how to deal with failures. A formal treatment of fault-tolerant, however, is important, since informal reasoning about failures is often based on hidden, implicit, assumptions and it is usually not clear whether all failures and combinations of failures have been covered. Moreover, given the complexity of the reasoning, it is clear that mechanical support is indispensable.

In this paper we use an assertional method to verify distributed protocols. That is, logical formulas (assertions) characterize the required service, the protocols, and assumptions about the underlying communication mechanism. Besides primitives to reason about timing, there are also primitives to express assumptions about the correctness of components and the failure hypothesis explicitly. To obtain mechanical support, we use the interactive proof checker PVS (Prototype Verification System) [ORS92,ORSvH95]. The PVS specification language is a higher-order typed logic and specifications can be structured into a hierarchy of (parameterized) theories. The tool contains a proof checker to construct proofs interactively and to rerun proofs automatically after small changes.

PVS is a very general tool with a rich specification language that can be used for a large number of applications. To guide the average user for a particular class of applications, it is convenient to have a clear sequence of steps to be performed and a number of re-usable theories and illustrative paradigms. Here we present our ideas in this direction on the verification of distributed protocols. The approach originates from previous manual verifications (e.g., for a distributed real-time arbitration protocol [Hoo94]). Mechanical support is obtained by defining general theories in PVS to reason about timing and failures.

To illustrate the approach, we consider a characteristic hierarchy of fault-tolerant protocols that should achieve a particular real-time computing service in spite of component failures. To achieve reliability, a group of servers is running on distinct processors. Processors may fail and servers should contain consistent service state information to redistribute workload. E.g., servers must agree on the identity of all correctly functioning processors, which is achieved by group membership services [Cri96]. As an example of such a protocol we consider a processor-group membership protocol of Cristian [Cri91] which is intended to achieve agreement on the identity of all correctly functioning processors in a dynamic network that can shrink with failures and grow with joins. We verify a basic version of the protocol, reasoning in terms of local clocks. For simplicity, we do not use the group identifiers mentioned in [Cri91] (assuming that the scheduling delay is zero). Further we assume fail-silent processors whereas in [Cri91] processors can suffer both crash and performance failures.

The membership algorithm uses a synchronous atomic broadcast service to communicate between processors. In [CASD95] it is shown that this broadcast service can indeed be implemented, using a lower-level point-to-point datagram service and internally synchronized local clocks. Here we indicate how to proceed with the verification of this atomic broadcast protocol in PVS. Interesting is the formulation of clock synchronization in terms of real-time, and the relation between reasoning with local clocks and real-time. Further, besides failures of processors, now also failures of links between processors are taken into account, re-using the theory that defines failure primitives. For details of the (manual) verification of the atomic broadcast protocol, we refer to [CASD95,ZH95].

In section 2 we define two general PVS theories for dealing with timing and failures. Our steps for protocol verification are presented in section 3. It is applied to a simple version of the membership protocol in section 4. In section 5 we show how the performance of this protocol can be improved, following [Cri91] more closely. A few basic ingredients of the verification of the underlying atomic broadcast protocol are presented in section 6. Finally, section 7 contains concluding remarks.

2 Basic Theories to deal with Timing and Failures

To reason about timing, we present below a general PVS theory *TimePrim* which has three parameters, a time domain *Time*, and two orders $<$ and $\leq$ on *Time* (the predicates *strict_order?* and *partial_order?* are pre-defined in the PVS prelude). Note that, instead of defining $\leq$, we add an assumption about the relation between the two orders. This makes it possible to import the theory with, e.g., the pre-defined relation $\leq$ on the reals and get more support from the decision procedures of the PVS proof checker.

We do not show all details of theory *TimePrim*, but indicate that intervals of time points are defined and that the boolean connectives are overloaded for predicates on *Time* (i.e., functions from *Time* to *bool*). Primitives are introduced to express that a predicate holds *inside* or *during* a set of time points (usually an interval). Finally we show a few simple lemmas, named *dur_inside* and *sub_dur*. All lemmas and theorems presented in this paper have been proved by means of the interactive proof checker of PVS. For instance, *dur_inside* can be proved automatically by one command, whereas *sub_dur* has been proved by eight commands.

$\mathrm{TimePrim}[\mathrm{Time}: \textsc{type}, < : (\mathrm{strict\_order?}[\mathrm{Time}]),$
$\qquad\qquad \leq : (\mathrm{partial\_order?}[\mathrm{Time}])] : \textsc{theory}$
$\quad \textsc{begin}$
$\quad \textsc{assuming}$
$\quad\quad \mathrm{leq\_less}: \textsc{assumption}\ \forall (t_1, t_2 : \mathrm{Time}) : t_1 \leq t_2 \Leftrightarrow t_1 < t_2 \vee t_1 = t_2$
$\quad\quad \textsc{endassuming}$
$\quad t, t_0, t_1, t_2, t_3, t_4 : \textsc{var}\ \mathrm{Time}$
$\quad [t_0, t_1] : \mathrm{setof}[\mathrm{Time}] = \{t \mid t_0 \leq t \wedge t \leq t_1\}$
$\quad [t_0, t_1) : \mathrm{setof}[\mathrm{Time}] = \{t \mid t_0 \leq t \wedge t < t_1\}$
$\quad \ldots$
$\quad P, Q : \textsc{var}\ \mathrm{pred}[\mathrm{Time}]$
$\quad \neg P : \mathrm{pred}[\mathrm{Time}] = \lambda\, t : \neg P(t);$
$\quad P \wedge Q : \mathrm{pred}[\mathrm{Time}] = \lambda\, t : P(t) \wedge Q(t);$
$\quad \ldots$
$\quad I : \textsc{var}\ \mathrm{setof}[\mathrm{Time}]$

$\quad P\ \mathbf{in}\ I : \mathrm{bool} = \exists\, t : t \in I \wedge P(t)$

$\quad P\ \mathbf{during}\ I : \mathrm{bool} = \forall\, t : t \in I \Rightarrow P(t)$

$\quad \mathrm{dur\_inside}: \textsc{lemma}\ \neg(P\ \mathbf{in}\ I) \Leftrightarrow (\neg P)\ \mathbf{during}\ I$

$\quad \mathrm{sub\_dur}: \textsc{lemma}$
$\quad\quad (\exists\, t_1, t_2 : P\ \mathbf{during}\ [t_1, t_2] \wedge t_1 \leq t_3 \wedge t_4 \leq t_2) \Rightarrow P\ \mathbf{during}\ [t_3, t_4]$
$\textsc{end}\ \mathrm{TimePrim}$

Next we present a theory to reason about failures of a certain type of *Resources*. To express the timing of failures, we again assume given a time domain with two orders and import theory *TimePrim*. Based on the primitives *Start* and *Crash*, it is defined when a resource is correct. We also show an example of a general property that has been proved, where *dichotomous?* expresses that for all t_1, t_2 we have $t_1 \leq t_2$ or $t_2 \leq t_1$.

$$
\begin{array}{l}
\mathrm{FailPrim}[\mathrm{Resources}: \ \textsc{type}, \mathrm{Time}: \ \textsc{type}, <: \ (\mathrm{strict\_order?}[\mathrm{Time}]), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \leq: \ (\mathrm{partial\_order?}[\mathrm{Time}])]: \ \textsc{theory} \\
\quad \textsc{begin} \\
\quad \textsc{assuming} \\
\quad\ \ \mathrm{leq\_less}: \ \textsc{assumption} \ \forall \ (t_1, t_2: \ \mathrm{Time}): \ t_1 \leq t_2 \Leftrightarrow t_1 \ < \ t_2 \vee t_1 = t_2 \\
\quad\ \ \textsc{endassuming} \\
\quad \textsc{importing} \ \mathrm{TimePrim}[\mathrm{Time}, <, \leq] \\
\quad r: \ \textsc{var} \ \mathrm{Resources} \\
\quad t, t_0, t_1, t_2: \ \textsc{var} \ \mathrm{Time} \\
\\
\quad \mathrm{Start}, \mathrm{Crash}: \ [\mathrm{Resources} \rightarrow \ \mathrm{pred}[\mathrm{Time}]] \\
\\
\quad \mathrm{Correct}(r)(t): \ \mathrm{bool} \ = \ \exists \ t_0: \ t_0 \leq t \wedge \mathrm{Start}(r)(t_0) \wedge (\neg \mathrm{Crash}(r)) \ \mathbf{during} \ [t_0, t] \\
\\
\quad \mathrm{Correct\_start}: \ \textsc{lemma} \\
\qquad \mathrm{Correct}(r) \ \mathbf{during} \ [t_1, t_2] \wedge t_1 \leq t_2 \wedge \mathrm{dichotomous?}[\mathrm{Time}](\leq) \Rightarrow \\
\qquad\ \ \exists t_0: \ t_0 \leq t_1 \wedge \mathrm{Start}(r)(t_0) \wedge \mathrm{Correct}(r) \ \mathbf{during} \ [t_0, t_2] \\
\ \textsc{end} \ \mathrm{FailPrim}
\end{array}
$$

3 Approach to Protocol Verification

To verify a distributed protocol for a network of processors, we perform the following steps.

1. Model the application domain; describe the main primitives that are needed to express the service specification.
2. Specify the service that should be provided by the network.
3. Specify the communication mechanism between processors.
4. Specify the protocol performed by each processor (or each type of processor if there are processors with different roles in the protocol). The specification of the protocol performed by a processor should only refer to the local state and the external interface of the processor itself.
5. Verify the protocol by proving that the conjunction of the specifications of the processors (point 4) and the communication mechanism (point 3) leads to the required service (point 2).

The last step, where the conjunction of the specifications is taken, is justified by earlier work on compositional proof rules for concurrency (see, e.g., [Hoo91]).

Note that for a fault-tolerant protocol the specifications also include the failure hypothesis, i.e., assumptions about the correctness of components and about the consequences of failures.

4 Membership Protocol

Following the steps of the previous section, we verify the basic part of a membership protocol. Consider a network of processors, where processors can crash and might be recovered. Each processor maintains a local view on the set of processors that are considered to be correct. Further each processor has a local boolean variable *Joined* which indicates whether a processor has correctly joined the network and its view can be inspected. Since it will in general be impossible to guarantee that these views are the same at any point in real-time, the specification of the membership service is expressed in terms of local clocks. It is required that the view of a correct and joined processor p at time T *on the local clock of* p equals the view of any other correct and joined processor q at time T *on the local clock of* q. This matches nicely with the specification of the underlying synchronous atomic broadcast service which is also expressed in terms of local clocks. To express these specifications, we first introduce in the next section primitives to refer to local clocks and the correctness of processors.

4.1 Modeling of the application domain

The application domain is modeled in theory *Config*, parameterized by the type of *Processors*. Since the aim is to reason in terms of local clocks, we define a discrete domain of clock values (using integers for simplicity) and import theory *TimePrim*. A local clock value is defined by a pair consisting of a clock value and a processor. We define orders $<$ and $\leq$ on local clock values and import theory *FailPrim* (and hence implicitly *TimePrim*). When theory *Config* is type-checked, PVS generates so called Type Check Conditions, corresponding to the type constraints on the parameters of *FailPrim* and the assuming clause. E.g., it is required that relation $<$ is indeed a strict order on local clock values.

Config[Processors : TYPE] : THEORY
  BEGIN
  ClockValues : TYPE = int
  IMPORTING TimePrim[ClockValues, $<$, $\leq$]

  $T, T_0, T_1, T_2, T_3, T_4$: VAR ClockValues
  NonNegClockValues : TYPE = $\{T \mid T \geq 0\}$

  LocalClockVals : TYPE = [ClockValues, Processors]
  LT, LT_1, LT_2 : VAR LocalClockVals
  $LT_1 < LT_2$: bool =
    $\mathrm{PROJ\_1}(LT_1) < \mathrm{PROJ\_1}(LT_2) \wedge \mathrm{PROJ\_2}(LT_1) = \mathrm{PROJ\_2}(LT_2)$;
  $LT_1 \leq LT_2$: bool = $LT_1 < LT_2 \vee LT_1 = LT_2$
  IMPORTING FailPrim[Processors, LocalClockVals, $<$, $\leq$]

By theory *FailPrim* we obtain primitives such as $Start(p)(T, p)$ to express that processor p starts at local clock value (T, p). To avoid duplication of processor p in this notation, we abbreviate it as *start*$(p)(T)$. Similarly for the other primitives. Since the membership protocol can be verified in terms of clock values, only the abbreviations *start*, *crash*, etc., are used. The use of processors in local clock values will become relevant for the implementation of atomic broadcast.

To express membership properties, we define the view of a processor p at a local time LT, representing the set of processors that are considered to be correct by p at LT. Predicate *Joined* expresses whether a processor has joined the group of processors and its view can be inspected. Again we define a few suitable abbreviations.

p, q, r : VAR Processors
start$(p)(T)$: bool = Start$(p)(T, p)$
crash$(p)(T)$: bool = Crash$(p)(T, p)$
correct$(p)(T)$: bool = Correct$(p)(T, p)$

View$(p)(LT)$: setof[Processors]
view$(p)(T)$: setof[Processors] = View$(p)(T, p)$

V : VAR [ClockValues $\rightarrow$ setof[Processors]]
$p \in V$: pred[ClockValues] = $\lambda\, T : p \in V(T)$

Joined$(p)(LT)$: bool
joined$(p)(T)$: bool = Joined$(p)(T, p)$
END Config

Henceforth, declarations of variables are not repeated, using for instance variable T ranging over *ClockValues*, p ranging over *Processors*, etc.

4.2 Specification of the required service

Theory *Membership* specifies the required membership service. Recall that we have introduced abbreviations to avoid duplication of processors, and hence property *Agree* below expresses that the view of p at T *measured on the clock of* p equals the view of q at T *measured on the clock of* q. Given that local clocks are synchronized, as will be expressed later, views do not differ too much in real-time.

Membership[Processors : TYPE] : THEORY
BEGIN
IMPORTING Config[Processors]

Agree : bool $= \forall\, T, p, q$:
correct$(p)(T) \wedge$ correct$(q)(T) \wedge$ joined$(p)(T) \wedge$ joined$(q)(T) \Rightarrow$
view$(p)(T) =$ view$(q)(T)$

Note that agreement allows arbitrary views; e.g., all sets could be empty. Following [Cri91] such trivial solutions are avoided by requiring that a joined processor is a member of its own view. Moreover, timeliness requirements are added to specify that a started processor is joined and a crashed processor is deleted from the view of a correct processor within a bounded amount of time.

Recognition : bool $= \forall\, T, p :\ \text{correct}(p)(T) \wedge \text{joined}(p)(T) \Rightarrow p \in \text{view}(p)(T)$

JoinDelay : NonNegClockValues

Join : bool $= \forall\, T_1, T_2, p :\ \text{start}(p)(T_1) \wedge \text{correct}(p)$ **during** $[T_1, T_2] \Rightarrow$
joined(p) **during** $[T_1\ +\ \text{JoinDelay}, T_2]$

DetectionDelay : ClockValues

Detect : bool $= \forall\, T_1, T_2, p, q :$
crash$(p)(T_1) \wedge \neg\text{correct}(p)$ **during** $[T_1, T_2] \wedge \text{correct}(q)$ **during** $[T_1, T_2] \Rightarrow$
$p \notin \text{view}(q)$ **during** $[T_1 + \text{DetectionDelay}, T_2]$

MembSpec : bool $=$ Agree $\wedge$ Recognition $\wedge$ Join $\wedge$ Detect
END Membership

To get more confidence in a specification, e.g. that it is not contradictory and indeed expresses the desired properties, it is advisable (see, for instance, [ORSvH95]) to try to prove some properties that ought to hold. Here we show, as an example, an alternative formulation of the join property.

JoinAlt : bool $= \forall\, T_1, T_2, p, q :$
start$(p)(T_1) \wedge \text{correct}(p)$ **during** $[T_1, T_2] \wedge \text{correct}(q)$ **during** $[T_1, T_2] \Rightarrow$
$p \in \text{view}(q)$ **during** $[T_1 + \text{JoinDelay}, T_2]$

join_equiv : LEMMA Agree $\wedge$ Recognition $\wedge$ Join $\Rightarrow$ JoinAlt

4.3 Specification of the underlying communication mechanism

To implement the membership service, it is assumed that processors can use an atomic broadcast service. We introduce primitives to express that a message can be broadcast by a processor and that it can be delivered at a processor. Again abbreviations are used to avoid duplication of processors and to reason with clock values. Next the properties of synchronous atomic broadcast are axiomatized. First we specify *Termination*, that is, a message that is broadcast by a correct processor at time T on its own clock is delivered by all correct processors at time $T{+}\mathit{BroadcastDelay}$ on their clocks.

Broadcast[Processors : TYPE, Messages : TYPE] : THEORY
BEGIN
IMPORTING Config[Processors]
m : VAR Messages
Broadcast$(p, m)(LT)$: bool
Deliver$(p, m)(LT)$: bool
broadcast$(p, m)(T)$: bool $=$ Broadcast$(p, m)(T, p)$
deliver$(p, m)(T)$: bool $=$ Deliver$(p, m)(T, p)$

BroadcastDelay : NonNegClockValues

$$
\begin{array}{l}
\text{Termination}: \ \text{bool} \ = \forall\ p, m, T: \\
\quad \text{broadcast}(p, m)(T) \wedge \text{correct}(p)\ \textbf{during}\ [T, T + \text{BroadcastDelay}] \Rightarrow \\
\quad\quad (\forall\ q: \text{correct}(q)\ \textbf{during}\ [T, T + \text{BroadcastDelay}] \Rightarrow \\
\quad\quad\quad\quad \text{deliver}(q, m)(T + \text{BroadcastDelay}))
\end{array}
$$

Note that *Termination* only specifies the delivery of a message if the processor that broadcasts it stays correct. To restrict the possible behavior in case of a processor crash during the broadcast action, we specify two atomicity properties: *Agreement* expresses that if a correct processor delivers a message at time T on its clock then this message is delivered by all correct processors at T on their clocks; *Integrity* specifies that if a message is delivered at time T then it was indeed broadcast at time $T - BroadcastDelay$. We have omitted an order property, since it is not relevant for the correctness of the membership protocol.

$$
\begin{array}{l}
\text{Agreement}: \ \text{bool} \ = \forall\ p, m, T: \\
\quad \text{deliver}(p, m)(T) \wedge \text{correct}(p)\ \textbf{during}\ [T - \text{BroadcastDelay}, T] \Rightarrow \\
\quad\quad (\forall\ r: \ \text{correct}(r)\ \textbf{during}\ [T - \text{BroadcastDelay}, T] \Rightarrow \text{deliver}(r, m)(T))
\end{array}
$$

$$
\begin{array}{l}
\text{Integrity}: \ \text{bool} \ = \forall\ p, m, T: \\
\quad \text{deliver}(p, m)(T) \wedge \text{correct}(p)\ \textbf{during}\ [T - \text{BroadcastDelay}, T] \Rightarrow \\
\quad\quad \exists\ q: \ \text{broadcast}(q, m)(T - \text{BroadcastDelay})
\end{array}
$$

$$
\text{AtomicBroadcast}: \ \text{bool} \ = \ \text{Termination} \wedge \text{Agreement} \wedge \text{Integrity}
$$

END Broadcast

4.4 Protocol specification

We specify a simple membership protocol where each correct processor p broadcasts a message *(p,present)* at least once every *PresentPeriod* time units, as expressed by *StartBcastPresent*.

MembProt[Processors : TYPE] : THEORY
BEGIN
IMPORTING Config[Processors]
MessageTypes : TYPE = {present}
Messages : TYPE = [Processors, MessageTypes]
IMPORTING Broadcast[Processors, Messages], Membership[Processors]

$$
\begin{array}{l}
\text{StartBcastPresent}: \ \text{bool} \ = \forall\ p, T_1, T_2: \text{start}(p)(T_1) \wedge \text{correct}(p)\ \textbf{during}\ [T_1, T_2] \Rightarrow \\
\quad \forall (T_3 \ \mid \ T_3 \in [T_1, T_2)): \\
\quad\quad \exists\ (T_4 \ \mid \ T_4 \in [T_1, T_3]): T_3 \ - \ \text{PresentPeriod} \leq T_4 \wedge \text{broadcast}(p, (p, \text{present}))(T_4)
\end{array}
$$

Using parameter *LocalJoinDelay*, we specify that a correct processor will be joined *LocalJoinDelay* time units after its start.

LocalJoinDelay : VAR NonNegClockValues

$$
\begin{array}{l}
\text{ProcJoin}(\text{LocalJoinDelay}): \ \text{bool} \ = \forall\ p, T_1: \text{joined}(p)(T_1) \wedge \text{correct}(p)(T_1) \Leftrightarrow \\
\quad \exists T_0: \ T_0 \ + \ \text{LocalJoinDelay} \leq T_1 \wedge \text{correct}(p)\ \textbf{during}\ [T_0, T_1]
\end{array}
$$

Next we specify that q is a member of the view of p if a *(q,present)* message has been received not longer than *PresentPeriod* time units ago (measured on the clock of p). Further only *present* messages after the start of p are used.

$$
\begin{array}{l}
\text{LocalView}: \text{bool} = \forall\, p, T_1, T_2 : \text{start}(p)(T_1) \wedge \text{correct}(p)\ \textbf{during}\ [T_1, T_2] \Rightarrow \\
\quad \forall (T_3 \mid T_3 \in [T_1, T_2]), q : \\
\quad\quad q \in \text{view}(p)(T_3) \Leftrightarrow \\
\quad\quad\quad \exists\, (T_4 \mid T_4 \in [T_1, T_3]) : \\
\quad\quad\quad\quad T_3 - \text{PresentPeriod} \leq T_4 \wedge \text{deliver}(p, (q, \text{present}))(T_4)
\end{array}
$$

Clearly processes should only broadcast their own identity. Moreover it is required that incorrect processors do not broadcast messages.

$$
\text{OwnIdentityOnly}: \text{bool} = \forall\, p, q, T : q \neq p \Rightarrow \neg\text{broadcast}(p, (q, \text{present}))(T)
$$

$$
\text{FailSilent}: \text{bool} = \forall\, p, m, T : \text{broadcast}(p, m)(T) \Rightarrow \text{correct}(p)(T)
$$

$$
\begin{array}{l}
\text{MembProtSpec}(\text{LocalJoinDelay}): \text{bool} = \text{StartBcastPresent} \wedge \\
\quad \text{ProcJoin}(\text{LocalJoinDelay}) \wedge \text{LocalView} \wedge \text{OwnIdentityOnly} \wedge \text{FailSilent} \\
\text{END MembProt}
\end{array}
$$

4.5 Protocol verification

Next the PVS interactive proof checker is used to prove that the specification of the membership protocol together with the properties of atomic broadcast lead to the membership service. As usual, some constraints on the timing constants have to hold. E.g., to obtain *Agree*, the *LocalJoinDelay* should be greater than the *PresentPeriod* plus the *BroadcastDelay*, because this is the time needed to be sure that all *present* messages have been received. Since the *LocalJoinDelay* should be smaller than the *Joindelay*, this also gives the bound on *JoinDelay*.

$$
\begin{array}{l}
\text{MembVerif[Processors : TYPE] : THEORY} \\
\quad \text{BEGIN} \\
\quad \text{IMPORTING MembProt[Processors]} \\
\\
\quad \text{agree\_verif : LEMMA LocalJoinDelay} \geq \text{PresentPeriod} + \text{BroadcastDelay} \wedge \\
\quad\quad \text{LocalView} \wedge \text{Agreement} \wedge \text{ProcJoin}(\text{LocalJoinDelay}) \Rightarrow \text{Agree} \\
\\
\quad \text{recogn\_verif : LEMMA LocalJoinDelay} \geq \text{BroadcastDelay} \wedge \\
\quad\quad \text{ProcJoin}(\text{LocalJoinDelay}) \wedge \text{StartBcastPresent} \wedge \text{LocalView} \wedge \text{Termination} \Rightarrow \\
\quad\ \ \text{Recognition} \\
\\
\quad \text{join\_verif : LEMMA LocalJoinDelay} \leq \text{JoinDelay} \wedge \text{ProcJoin}(\text{LocalJoinDelay}) \Rightarrow \text{Join} \\
\\
\quad \text{detect\_verif : LEMMA DetectionDelay} \geq \text{BroadcastDelay} + \text{PresentPeriod} \wedge \\
\quad\quad \text{LocalView} \wedge \text{OwnIdentityOnly} \wedge \text{FailSilent} \wedge \text{Integrity} \Rightarrow \text{Detect} \\
\\
\quad \text{memb\_correct : THEOREM JoinDelay} \geq \text{PresentPeriod} + \text{BroadcastDelay} \wedge \\
\quad\quad \text{DetectionDelay} \geq \text{BroadcastDelay} + \text{PresentPeriod} \wedge \\
\quad\quad\quad \text{MembProtSpec}(\text{JoinDelay}) \wedge \text{AtomicBroadcast} \Rightarrow \text{MembSpec} \\
\quad \text{END MembVerif}
\end{array}
$$

Not shown are seven auxiliary lemmas used to structure the proof. On the average, the proofs of lemmas and theorems required around twenty user interactions.

5 Improving the join delay

To improve the join delay, [Cri91] introduces *(p,new)* messages to indicate the start of processor p. Property *StartUp* expresses that a processor broadcasts a *new* message when it starts. A started processor receives its own *new* message and then, as long as it is correct, periodically broadcast a *present* message. Further all correct processors respond to a *new* message by immediately broadcasting a *present* message. Property *BroadcastReason* expresses that there are no other *present* messages, so all correct processors broadcast at the same local clock value.

$$
\begin{array}{l}
\text{PeriodProt}[\text{Processors}: \textsc{type}]: \textsc{theory} \\
\quad \textsc{begin} \\
\quad \textsc{importing}\ \text{Config}[\text{Processors}] \\
\quad \text{MessageTypes}: \textsc{type} \ = \{\text{new}, \text{present}\} \\
\quad \text{Messages}: \textsc{type} \ = [\text{Processors}, \text{MessageTypes}] \\
\quad \textsc{importing}\ \text{Broadcast}[\text{Processors}, \text{Messages}],\ \text{Membership}[\text{Processors}] \\
\\
\quad \text{StartUp}: \text{bool} \ = \forall\ p, T:\ \text{start}(p)(T) \wedge \text{correct}(p)(T) \Leftrightarrow \text{broadcast}(p, (p, \text{new}))(T) \\
\\
\quad \text{deliver\_new}(p)(T): \text{bool} \ = \ \exists\ q:\ \text{deliver}(p, (q, \text{new}))(T) \\
\quad \text{deliver\_present}(p)(T): \text{bool} \ = \ \exists\ q:\ \text{deliver}(p, (q, \text{present}))(T) \\
\\
\quad \text{PresentPeriod}: \text{NonNegClockValues} \\
\quad \text{PeriodPoint}(T_1, T_3): \text{bool} \ = \ \exists\ (k:\ \text{nat}):\ T_3 = T_1 \ + \ k \times \text{PresentPeriod} \\
\\
\quad \text{BroadcastPresent}: \text{bool} \ = \forall\ p, q, T_1, T_2, T_3: \\
\quad\quad \text{deliver}(p, (q, \text{new}))(T_1) \wedge \text{correct}(p)\ \textbf{during}\ [T_1, T_2] \wedge \\
\quad\quad\quad \neg \text{deliver\_new}(p)\ \textbf{during}\ (T_1, T_2] \wedge T_3 \in [T_1, T_2] \wedge \text{PeriodPoint}(T_1, T_3) \Rightarrow \\
\quad\quad \text{broadcast}(p, (p, \text{present}))(T_3) \\
\\
\quad \text{BroadcastReason}: \text{bool} \ = \forall\ p, T:\ \text{broadcast}(p, (p, \text{present}))(T) \Rightarrow \\
\quad\quad \exists q, T_0: T_0 \leq T \wedge \text{deliver}(p, (q, \text{new}))(T_0) \wedge \\
\quad\quad\quad \text{correct}(p)\ \textbf{during}\ [T_0 \ - \ \text{BroadcastDelay}, T] \wedge \\
\quad\quad\quad\quad \neg \text{deliver\_new}(p)\ \textbf{during}\ (T_0, T] \wedge \text{PeriodPoint}(T_0, T)
\end{array}
$$

Next we specify that the local view corresponds to the last delivered *present* messages and that a processor becomes joined when it receives its own *present* message. As before, we have *OwnIdentityOnly* and *FailSilent*.

$$
\begin{array}{l}
\quad \text{lastdelivered}(p, q)(T): \text{bool} \ = \exists\ T_0:\ T_0 \leq T \wedge \text{deliver}(p, (q, \text{present}))(T_0) \wedge \\
\quad\quad \neg \text{deliver\_present}(p)\ \textbf{during}\ (T_0, T] \wedge \text{correct}(p)\ \textbf{during}\ [T_0 \ - \ \text{BroadcastDelay}, T] \\
\\
\quad \text{LocalView}: \text{bool} \ = \forall\ p, T:\ \text{correct}(p)(T) \Rightarrow \\
\quad\quad (\forall\ q:\ q \in \text{view}(p)(T) \Leftrightarrow \text{lastdelivered}(p, q)(T)) \\
\\
\quad \text{ProcJoin}: \text{bool} \ = \forall\ p, T_1:\ \text{joined}(p)(T_1) \wedge \text{correct}(p)(T_1) \Leftrightarrow \\
\quad\quad \exists T_0: T_0 \leq T_1 \wedge \text{deliver}(p, (p, \text{present}))(T_0) \wedge \\
\quad\quad\quad \text{correct}(p)\ \textbf{during}\ [T_0 \ - \ \text{BroadcastDelay}, T_1]
\end{array}
$$

Then again the membership properties can be verified. Interesting is that now the *JoinDelay* should be greater or equal than two times *BroadcastDelay*, because a new processor first sends a *new* message; all correct processors receive this message after *BroadcastDelay*, and then immediately send a *present* message, which is received by any correct processor after another period of *BroadcastDelay* time units.

$$\begin{array}{l} \text{agree\_verif}: \text{LEMMA} \\ \quad \text{LocalView} \wedge \text{Agreement} \wedge \text{ProcJoin} \wedge \text{Integrity} \wedge \text{OwnIdentityOnly} \wedge \\ \qquad \text{BroadcastReason} \wedge \text{StartUp} \wedge \text{Termination} \wedge \text{BroadcastPresent} \Rightarrow \text{Agree} \end{array}$$

$$\begin{array}{l} \text{join\_verif}: \text{LEMMA JoinDelay} \geq 2 \times \text{BroadcastDelay} \wedge \\ \quad \text{ProcJoin} \wedge \text{StartUp} \wedge \text{Termination} \wedge \text{BroadcastPresent} \Rightarrow \text{Join} \end{array}$$

6 Implementation of Atomic Broadcast

An implementation of the atomic broadcast service assumed in the previous section can be found in [CASD95]. To verify this protocol, we can again follow the steps of section 3, now using the theory of section 4.3 as a specification of the required service. Here we only show a few basic theories for such a verification in PVS, indicating how to deal with different notions of time and failing links between processors.

6.1 Local clock synchronization

Recall that the properties of atomic broadcast express the timing of actions of a processor in terms of the local clock of the processor itself. For instance, a broadcast action on processor p at local time T, measured on the clock of p, leads to a deliver action on another processor q at local time $T+BroadcastDelay$, measured on the clock of q. To be able to verify a protocol which realizes this on top of a simple point-to-point datagram service, we have to make some assumption about the relation between local clocks. A common assumption is that local clocks of correct processors are *internally synchronized*, that is, at any point in real-time they differ at most ϵ. To formulate this in PVS we define a domain *RealTime*, equal to the real numbers, and postulate for each processor p the existence of a surjective function $C(p)$ from *RealTime* to *ClockValues*.

$$\begin{array}{l} \text{LocalClockSynchr[Processors : TYPE] : THEORY} \\ \quad \text{BEGIN} \\ \quad \text{IMPORTING Config[Processors]} \\ \quad \text{RealTime : TYPE} \;=\; \text{real} \\ \quad C(p): \;(\text{surjective?[RealTime, ClockValues]}) \end{array}$$

Since we would like to express that predicates (e.g., about the correctness of processors) hold at certain points in real-time, we define a function rt which converts a predicate P on local clock values to one on real-time. By postulating this as a *conversion* in PVS, we can write $P(t)$, for $t \in RealTime$, which is interpreted as $rt(P)(t)$. Using surjectivity of $C(p)$ we can show that if a predicate holds on a local clock value, then it also holds at a point in real-time. To obtain the other direction we have to introduce axiom *real_to_local*. This leads to the desired relation between local clock values and real-time, as expressed by lemma *local_real_conv*.

t : VAR RealTime
P : VAR pred[LocalClockVals]
$\mathrm{rt}(P)(t)$: bool $= \exists\, p : P(C(p)(t), p)$
CONVERSION rt
local_to_real : LEMMA $P(T, p) \Rightarrow \exists\, t : C(p)(t) = T \wedge P(t)$
real_to_local : AXIOM $P(t) \Rightarrow P(C(p)(t), p)$
local_real_conv : LEMMA $P(T, p) \Leftrightarrow \exists\, t : C(p)(t) = T \wedge P(t)$

Next we formulate the usual axiom expressing internal clock synchronization, where *abs* yields the absolute value. Note that $Correct(p)(t)$ is interpreted by means of conversion rt.

ε : ClockValues
local_clock_synch : AXIOM
$\mathrm{Correct}(p)(t) \wedge \mathrm{Correct}(q)(t) \Rightarrow \mathrm{abs}(C(p)(t) - C(q)(t)) < \varepsilon$

Since the specification of the atomic broadcast service is written in terms of local clock values, the aim is to verify the protocol using local clock values only, avoiding real-time. Important part of such a proof is that we can transform a property in terms of the local clock of one processor into a property in terms of the local clock of another processor. Therefore we have proved the following two theorems which are based on the internal clock synchronization axiom, but do not refer to real-time.

clock_synchr : THEOREM
$P(T, p) \wedge \mathrm{Correct}(p)(T, p) \wedge \mathrm{Correct}(q)(T, p) \Rightarrow P$ **in** $[(T - \varepsilon, q), (T + \varepsilon, q)]$

clock_change : THEOREM
P **in** $[(T_1, p), (T_2, p)] \wedge$
$\mathrm{Correct}(p)$ **during** $[(T_1, p), (T_2, p)] \wedge \mathrm{Correct}(q)$ **during** $[(T_1, p), (T_2, p)] \Rightarrow$
P **in** $[(T_1 - \varepsilon, q), (T_2 + \varepsilon, q)]$
END LocalClockSynchr

6.2 Specification of datagram service

The atomic broadcast protocol of [CASD95] is based on a datagram service which allows the transmission of messages along links between pairs of nodes. Corresponding to step 3 of our approach, we specify this communication mechanism in theory *Datagram*, with a type *Links* as one of the parameters. Since links might crash, we import theory *FailPrim*, now with *Links* as a parameter. Assume given a function *links* which gives the set of links of a processor. We define primitives to express that a processor p starts sending or receiving a message m along a link l at local clock value LT, together with the usual abbreviations for clock values. Note that we require that link l is member of the links of p.

$\text{Datagram}[\text{Processors}: \textsc{type}, \text{Links}: \textsc{type}, \text{Messages}: \textsc{type}]: \textsc{theory}$
$\quad \textsc{begin}$
$\quad \textsc{importing}\ \text{LocalClockSynchr}[\text{Processors}],\ \text{FailPrim}[\text{Links}, \text{LocalClockVals}, <, \leq]$
$\quad l: \textsc{var}\ \text{Links}$

$\quad \text{links}(p): \text{setof}[\text{Links}]$

$\quad \text{Send}(p, m, (l \mid l \in \text{links}(p)))(LT): \text{bool}$
$\quad \text{Receive}(p, m, (l \mid l \in \text{links}(p)))(LT): \text{bool}$
$\quad \text{send}(p, m, (l \mid l \in \text{links}(p)))(T): \text{bool} = \text{Send}(p, m, l)(T, p)$
$\quad \text{receive}(p, m, (l \mid l \in \text{links}(p)))(T): \text{bool} = \text{Receive}(p, m, l)(T, p)$

Property *BoundedCommunication* below expresses that, for a correct link l, any message that has been sent by p will be received by any processor q that is connected to p via l, between lower bound L and upper bound U. Following [CASD95], these time bounds are expressed in terms of clock values measured on the clock of any arbitrary correct processor r.

$\quad L, U: \text{NonNegClockValues}$
$\quad \text{LleqU}: \textsc{axiom}\ L \leq U$

$\quad \text{BoundedCommunication}: \text{bool} = \forall\ p, q, r, l, T, m:$
$\quad\ l \in \text{links}(p) \wedge l \in \text{links}(q) \wedge$
$\quad\quad \text{correct}(q)\ \textbf{during}\ [T+L, T+U] \wedge \text{correct}(r)\ \textbf{during}\ [T+L, T+U] \wedge$
$\quad\quad\ \text{Correct}(l)\ \textbf{during}\ [(T, r), (T+U, r)] \wedge$
$\quad\quad\quad \text{Send}(p, m, l)(T, r) \Rightarrow$
$\quad\quad\quad\ \text{Receive}(q, m, l)\ \textbf{in}\ [(T+L, r), (T+U, r)]$

When a link fails, there are several possible failure assumptions. For instance, messages might get lost (omission failures), messages might be received outside the specified time bounds (timing failures), messages might be corrupted, or spontaneous messages might be generated. To prove correctness of the atomic broadcast protocol, it is assumed that only omission failures occur.

$\quad \text{OnlyOmissionFailures}: \text{bool} = \forall\ q, r, l, T, m:$
$\quad\ l \in \text{links}(q) \wedge \text{correct}(r)(T) \wedge \text{Receive}(q, m, l)(T, r) \Rightarrow$
$\quad\quad \exists\ p: p \neq q \wedge l \in \text{links}(p) \wedge \text{Send}(p, m, l)\ \textbf{in}\ [(T-U, r), (T-L, r)]$

$\quad \text{Datagram}: \text{bool} = \text{BoundedCommunication} \wedge \text{OnlyOmissionFailures}$
$\textsc{end}\ \text{Datagram}$

The next step in the verification is the formulation of properties representing the protocol of [CASD95]. Finally, it can be proved that this protocol and the specification of the datagram service lead to the atomic broadcast properties specified before. The proof proceeds by induction on the distance between the sender and the receiver, assuming that the network remains connected.

7 Concluding Remarks

An assertional method for the verification of distributed real-time and fault-tolerant protocols has been presented. To obtain mechanical support, two general theories concerning timing and failures have been defined for the verification system PVS. Our approach has been illustrated by the verification of a membership protocol, reasoning in terms of local clocks only. We also showed some basic theories for the verification of an atomic broadcast protocol to indicate, for instance, how to deal with real-time and local clock synchronization in relation to local clock reasoning. Note that failures of processors and links have been treated in a uniform way.

A good overview of related mechanical verifications can be found in [ORSvH95], where mainly applications related to aircraft flight control and their influence on the design of PVS are described. Relevant is also an extensive treatment of clock synchronization protocols in PVS by Shankar [Sha92]. A classical example of a hierarchical verification is the "CLI short stack", which has been verified using Nqthm [BHMY89].

Most of the literature mentioned above does not give a clear methodology that can easily be used by others. Sometimes the use of a formal framework structures the specification and the verification; see for instance the use of TLA for a treatment of the Byzantine generals problem [LM94]. Interesting in [AH97] is the formulation of a template for specifying Lynch-Vaandrager timed automata in PVS. Generic PVS theories to verify well-known algorithmic techniques such as divide-and-conquer have been formulated by Dold [Dol95]. Specific algorithms can be obtained by adding details to a general scheme.

To make our method more accessible, the aim is to formulate a general template of theories that has to be completed by the verifier of a particular protocol. In future work we also intend to work on the formulation of general proof principles that can be used to prove the correctness of classes of protocols. For instance, for protocols that are based on information diffusion, a general diffusion induction principle [CASD95] could be formulated in PVS. Closely related to our aim is an interesting recent paper by Rushby [Rus97] where a general transformation from an untimed synchronous system into a time-triggered implementation of fault-tolerant algorithms is presented. It provides a useful methodology for a particular class of applications

Acknowledgments

This paper has been inspired by earlier work with Ed de Gast and Ping Zhou on the manual verification of the protocols considered here. Many thanks goes to Flaviu Cristian for valuable and stimulating comments on this earlier work and on a previous version of the current paper.

References

[AH97] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, pages 171–185. LNCS 1201, Springer-Verlag, 1997.

[BHMY89] W.R. Brevier, W.A. Hunt, J.S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[CASD95] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118:158–179, 1995.

[Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[Cri96] F. Cristian. On the semantics of group communication. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 1–21. LNCS 1135, Springer-Verlag, 1996.

[Dol95] A. Dold. Representing, verifying and applying software development steps using the PVS system. In *Algebraic Methodology and Software Technology (AMAST'95)*, pages 431–444. LNCS 936, Springer-Verlag, 1995.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*. LNCS 558, Springer-Verlag, 1991.

[Hoo94] J. Hooman. Compositional verification of a distributed real-time arbitration protocol. *Real-Time Systems*, 6(2):173–205, 1994.

[JP96] B. Jonsson and J. Parrow, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 1135. Springer-Verlag, 1996.

[LM94] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 41–76. LNCS 863, 1994.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Rus97] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In C. Meadows and W. Sanders, editors, *Dependable Computing for Critical Applications 6*, pages 191–210, 1997.

[Sha92] N. Shankar. Mechanical verification of a generalized proctocol for byzantine fault tolerant clock synchronization. In *Proceedings Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 217–236. LNCS 571, Springer-Verlag, 1992.

[ZH95] P. Zhou and J. Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9(2):119–145, 1995.