Mechanical Verification of a Non-Blocking Atomic Commitment Protocol

Dmitri Chkliaev Peter van der Stok
Dept. of Computing Science
Eindhoven University of Technology
The Netherlands
e-mail: {dmitri,wsstok}@win.tue.nl

Jozef Hooman
Computing Science Institute
University of Nijmegen
The Netherlands
e-mail: hooman@cs.kun.nl

Abstract

This paper concerns the formal specification and mechanical verification of atomic commitment protocols (ACP's) for distributed database systems. In particular, we consider the non-blocking ACP of Babaoglu and Toueg, combined with our own termination protocol for recovered participants. A new method to specify such protocols has been developed. In this method, timed state machines are used to specify the processes, whereas the communication mechanism between processes is defined using assertions. All safety and liveness properties, including a new improved termination property, have been proved with the interactive proof checker of PVS. We also show that the original termination protocol of Babaoglu and Toueg has an error.

1. Introduction

When a transaction updates data at multiple sites, *atomicity* must be ensured, which means that all or none of the updates to the data are made permanent. In order to achieve atomicity, all participants of a transaction execute an *atomic commitment protocol* (ACP). The aim of ACP's is to reach a consistent final decision on the outcome of a transaction: commit or abort.

An ACP is said to be *non-blocking* if it allows a participant to terminate a transaction despite failures of other participants. Such protocols are desirable since they avoid a waste of resources at correct participants. Unfortunately the most popular ACP, the Two-Phase Commit (2PC) protocol, may lead to *blocking* executions where a transaction remains unterminated at some correct participants for any arbitrary period of time.

Non-blocking ACP's are usually combined with *termination protocols* for participants that crashed and then became operational again after being repaired. Their aim is to help a participant to terminate a transaction even despite the loss of some information in a crash.

ACP's are often used in safety-critical applications and therefore it is very important to guarantee their correctness. One method to increase the confidence in the correctness of ACP's is their formal verification. This, however, is quite challenging. By their definition, ACP's are distributed, fault-tolerant, and often timed. They usually involve two types of processes: participants and a coordinator, each performing different protocols. To increase fault-tolerance, they may use a complex communication mechanism such as reliable broadcast. Moreover, there are several complicated correctness criteria. For instance, in the protocol considered here, there are 6 very different correctness properties: 3 safety and 3 liveness properties. All these factors lead to a significant overall complexity of the verification of ACP's. Therefore it is not surprising that not much work has been done on their formal verification.

To study the verification of ACP's, we define a formal framework for their specification and verification, choose tool support, and select a particular protocol as a case study. Therefore, in the remainder of this introduction we address the following questions: 1) What protocol is chosen for this particular case study, and why? 2) How to specify this protocol? 3) How to obtain mechanical support for specification and verification? 4) What are the results of our verification efforts? 5) What are the main lessons learned from this study?

1) Protocols studied. It is well-known that if communication failures in a distributed system are possible, then every ACP may cause processes to become blocked. For systems without such failures, a number of non-blocking protocols has been developed in the early eighties, for instance the Three-Phase Commit (3PC) protocol [BHG87]. These protocols have a much higher time and message complexity than 2PC. They are also much more complex to program and to understand, because they have to consider a large number of possible system states and often depend on complex and costly sub-protocols for leader election and determining the last process to fail.

A more recent solution to the problem has been developed by Babaoglu and Toueg [BT93a]. They suggested a few non-blocking protocols that share the basic structure of 2PC and achieve the non-blocking property by exploiting the properties of a *uniform reliable broadcast* mechanism which is used to disseminate certain messages. The protocols differ only in the implementation of this broadcast algorithm and non-blocking is obtained without adding much complexity to that of 2PC. This modular approach results in protocols that are easy to understand and that don't require any additional sub-protocols. Furthermore, it is shown in [BT93a] that the best implementation of reliable broadcast makes their protocol as efficient as 2PC.

In [BT93b], Babaoglu and Toueg also suggested a termination protocol to be used in combination with their commitment protocol (they call it "recovery protocol" instead, but in our opinion this contradicts both the existing literature, see for instance [BHG87], and the nature of the protocol). We found an error in their protocol; there is a combination of crashes and message delays leading to inconsistent decisions. Instead of just correcting the error, we developed our own termination protocol for the ACP of Babaoglu and Toueg. Our protocol is simpler in some respects, it sometimes avoids an unnecessary decision to abort a transaction and has a new improved termination property.

In this paper, we consider the non-blocking ACP of Babaoglu and Toueg (ACP-BT), combined with our own termination protocol for recovered participants. There are two main reasons for choosing this protocol. First, the protocol seems to be a promising and practical solution to the problem of non-blocking atomic commitment, especially because a growing proportion of fault-tolerant distributed systems has a built-in support for broadcast services. Second, a non-trivial communication mechanism such as reliable broadcast presents an interesting challenge for existing specification and verification techniques. Some work has been done already on the verification of atomic broadcast protocols [ZH95] and higher-level fault-tolerant protocols that use broadcast services (for instance [Hoo97]), but apparently there is still a long way to go.

2) Specification method. We developed a new specification method, in which the processes are specified by means of timed state machines, whereas the communication mechanism between processes is specified using assertions. There are good reasons to combine state machines and assertions. First, the pseudo code description in [BT93a] of a process in ACP can be naturally transformed into a formal description in terms of a state machine. Second, reliable broadcast services allow many different implementations and assertional specifications concentrate on *properties* of reliable broadcast, without having to chose a particular implementation. Also here the formalization is close to the informal statement of these properties in [BT93a]. In this way, we try to avoid errors in the important step from informal to formal description. We successfully solved the problem of formally combining these two techniques and have obtained a specification that is very close to the original description of the protocol.

3) Mechanical support. The structure of ACP's and their correctness properties, as well as the complexity of reasoning about them, make completely automatic verification not feasible. This is why we use a higher-order interactive theorem prover. We have chosen PVS [PVS], because it has a convenient specification language and is relatively easy to learn and to use. The specification language of PVS is a strongly-typed higher-order logic. Specifications can be structured into a hierarchy of parameterized theories. The PVS system contains an interactive proof checker, which allows users to construct proofs interactively and to check proofs mechanically.

4) Verification results. All 6 correctness properties have been completely proved with the interactive proof checker of PVS. The protocol has been verified in an incremental fashion: first we verified the basic ACP-BT (without recovery), and then reused almost all of its proofs for the verification of the complete protocol. This verification was very complex and took several months to complete, but greatly improved our understanding of the protocol. PVS provided great help for managing and reusing the proofs. It is clear that without suitable tool support it would be almost impossible to complete a formal proof of such size and complexity, consisting of more then two hundred lemmas.

5) Lessons learned. In this study, the use of PVS helped to achieve more than just proving the correctness of a known protocol. First, we could find an error in the termination protocol of Babaoglu and Toueg only after we spent several weeks verifying our own termination protocol. Without PVS, we would probably not get sufficient insight for this. PVS also helped to formalize the correctness properties much better. For instance, the termination property for ACP's with recovery is defined in [BT93a] as: "If all participants that know about the transaction remain operational long enough, then they all decide". This is very vague, especially because it is not specified how long these participants should remain operational. Using PVS, we developed a new termination property for our protocol, which specifies timing very precisely and expresses a higher degree of fault-tolerance.

Structure of the paper. In section 2, we present an informal description of the protocol and its correctness properties. We also explain the error in the termination protocol of Babaoglu and Toueg. Section 3 describes the specification of our framework in PVS, including our formalization of the termination property. Our approach to verification and some statistics on the verification efforts can be found in section 4. Section 5 contains concluding remarks.

2. The protocol and its correctness properties

We describe the protocols informally, using many definitions from [BT93a].

The model of system and transactions. The distributed system consists of a set of sites interconnected through a communication network. Sites can communicate with each other by exchanging messages. The model of computation is *synchronous* in the sense that there are known bounds for both relative speeds of processes and message delays.

At any given time, a process may be either *operational* or *crashed*. When operational, it follows exactly the actions specified by the program it is executing. Failures may cause operational processes to *crash*, after which they take no actions at all. A process that has never crashed since the start of the protocol is said to be *correct*. Crashed process may *recover* to become operational again. The only information available to a process during recovery is the contents of *stable storage*, because all other information is lost in a crash. The only type of stable storage used in ACP's is a *distributed transaction log* (DT-log).

A distributed transaction updates data at multiple sites, which are called *participants* of this transaction. We consider only one transaction at a time. One of the participants also acts as a *coordinator* to orchestrate the decision process about a transaction among participants.

General structure of ACP's. A Two-Phase ACP usually works as follows. In Phase 1, after receiving an order from the coordinator to *vote*, each participant sends to the coordinator its *vote*: YES or NO. A YES vote indicates that the participant is willing and able to make the updates to the data permanent. A NO vote indicates that for some reason (e.g., deadlock, concurrency control conflict) the participant is forced to discard the changes to the data. The coordinator collects all votes and makes a *decision*. If a YES vote was received from all participants, then the decision is **commit**; otherwise it is **abort**.

In Phase 2, the coordinator disseminates the decision to all participants. If a participant receives the decision from the coordinator, it decides **commit** or **abort** according to this decision; if no decision arrives, it should find some other way to reach a decision. After deciding **commit** a participant makes the transaction's updates permanent, after deciding **abort** it discards all the updates.

Correctness properties. It is clear that any reasonable ACP should satisfy at least the following properties (which can also be taken as a definition of an atomic commitment protocol):

AC1: All participants that decide reach the same decision.
AC2: If any participant decides **commit**, then all participants must have voted YES.
AC3: If all participants vote YES and no failures occur, then all participants decide **commit**.
AC4: Each participant decides at most once (that is, a decision is irreversible).

To disseminate the decision message to all participants, the coordinator often uses some *broadcast service*. This service may be described as *reliable* or *unreliable*. If the broadcast service is unreliable and the broadcaster crashes during the broadcast to a group of processes, then some processes (correct or not) from this group may deliver the message, whereas some other correct process may never deliver it. If the broadcast service is reliable, this scenario is impossible. It is easy to show, that if the coordinator uses unreliable broadcast in Phase 2 of some Two-Phase ACP, then this ACP cannot prevent blocking executions. However, the use of reliable broadcast may make the protocol *non-blocking*. More formally, an ACP is said to be non-blocking if it satisfies the following property[1] in addition to AC1-AC4:

AC5: Every *correct* participant eventually decides.

Note that this non-blocking property is stated in terms of participants that never crash. If a participant crashes and then recovers, it should reach a decision using a *termination protocol* rather then a basic ACP. When some recovered participant p cannot achieve a decision independently (usually because of the risk of violating AC1), it must contact other participants, ask them about their state, and then decide on the basis of their replies. In some scenarios, p will need the replies of *all other* participants in order to decide. In these scenarios, it is necessary that all other participants remain operational long enough to reply to p's request for help. A termination protocol is non-blocking, if this condition is also *sufficient* for p to recover. This means, if p recovers and stays operational for some sufficiently long period of time and all other participants also stay operational during this period of time, then p can always decide on the basis of their replies. It is easy to see that this *termination property* can also be defined in the following equivalent form:

AC6: If all participants remain operational long enough, then they all decide.

[1]Unlike [BT93a], we assume for simplicity that all participants of a transaction know about this transaction and they all execute an ACP.

The termination property for our protocol is an improvement of AC6. It is presented (with precise timing conditions) in section 3.

The ACP-BT protocol. Due to space limitations, we only explain the main differences between ACP-BT and "standard" Two-Phase ACP's, without getting into details of timing and message processing. The protocols, performed by each participant and the coordinator, are graphically presented in Fig. 1 and Fig. 2 respectively. These figures are extracted manually from our PVS specifications of the corresponding state machines and include our own termination protocol.

If a message has been disseminated to some group of processes *without* using reliable broadcast, then this action is called `send`, and a corresponding action at the destination is called `receive`. Each message is received within δ time units after being sent. If we use a reliable broadcast mechanism, the two actions are called `broadcast` and `deliver` respectively. Each message is delivered within Δ_b time units after being broadcasted and we have $\Delta_b \geq \delta$.

A participant has to vote when it receives a `vote_request` message from the coordinator. If it votes NO, it can decide immediately by moving into state `Aborted`. If it votes YES, it waits for the coordinator's decision in state `Waiting`. The coordinator decides as already explained, see Fig. 2. If a participant then delivers a **commit** or an **abort** decision, it moves to state `Committed` or `Aborted`, respectively. If no decision arrives before an appropriately chosen time-out expires, a participant can still decide **abort**, without becoming blocked!

The termination protocol. The first difference between our termination protocol and "standard" termination protocols is a higher level of abstraction in representing the DT-log. Without considering the tedious details of how the log is actually stored and managed, we just assume that a recovered process remembers the name of the state in which it crashed (e.g., `Committed`, `Aborted`, `Waiting` etc.), as well as the values of some variables that are especially important for reaching a decision. Of course, to implement a protocol specified in such a high-level fashion, we must combine it with a suitable log-management protocol, responsible for putting some values into the stable storage. In our opinion, this incremental approach leads to termination protocols that are much easier to represent, to understand, and to verify.

If some participant[2] p crashes in state `Committed` or `Aborted`, it recovers to the same state, thus completing termination. If it crashes in state `Waiting`, it cannot decide independently, so it starts the *requester's component* of the protocol by sending `help` message to all other participants. If some other participant q is operational (i.e. is not in one of the "crashed" states), it replies to this message according to the *responder's component* by sending `reply_commit` if in state `Committed`, `reply_abort` if in state `Aborted`, `reply_willing` if also in state `Willing`, and nothing otherwise.

If p then receives `reply_commit` or `reply_abort`, it moves to state `Committed` or `Aborted` respectively. If it receives `reply_willing`, it writes this reply into the log. If all other participants send `reply_willing` to p (which implies that the most malicious *total failure* has occurred, i.e. all participants crashed while waiting for the coordinator's decision), it decides **commit**. This is different from the termination protocol of Babaoglu and Toueg where, if all participants vote YES and then crash while waiting for the **commit** decision, they all decide **abort** during termination; the result is an unnecessary decision to abort a transaction. If p does not decide within $2 * \delta$ time units after sending the request for help, it repeats the request.

Error found. The responder's component of the termination protocol of Babaoglu and Toueg makes no distinction between those participants that voted YES and are still waiting for coordinator's decision, and those that recovered and are willing to commit. All such participants send `reply_willing` (in our notation) to a request for help. We show that this may lead to inconsistent decisions.

Consider the following scenario, in which the coordinator is correct and only one participant crashes. Suppose all participants send YES votes to the coordinator at real time $t1$. Also suppose that the simplest implementation of reliable broadcast is chosen (UTRB1 in [BT93a]), in which a) to broadcast a message, the coordinator sequentially sends it to each participant; b) each participant relays a message to all others before delivering it. In this implementation, each operational participant will deliver the **commit** decision by real time $t1 + 2 * \delta$. Suppose all of them deliver the decision at real time $t1 + 1.75 * \delta$.

Suppose now that some participant p crashes shortly after sending a YES vote and recovers by real time $t1 + 0.5 * \delta$. Then p immediately starts its termination protocol by sending `help` messages to all other participants. Suppose all of them receive this message by real time $t1 + \delta$. According to the responder's component, they all send `reply_willing`. Suppose p receives all these replies by real time $t1 + 1.5 * \delta$. Therefore p decides **abort** by real time $t1 + 1.5 * \delta$ (remember the difference with our protocol, where the decision would be **commit**). After that, all other participants deliver the **commit** decision at real time $t1 + 1.75 * \delta$ and they all decide **commit**. Thus p decides **abort** and all other participants decide **commit**. The proof of correctness in [BT93b] ignores such scenarios.

[2]Like [BT93b], we assume that the coordinator does not recover from crashes (for simplicity reasons, and also because the coordinator is not responsible for making any updates; it is the task of a participant located at the coordinator's site).

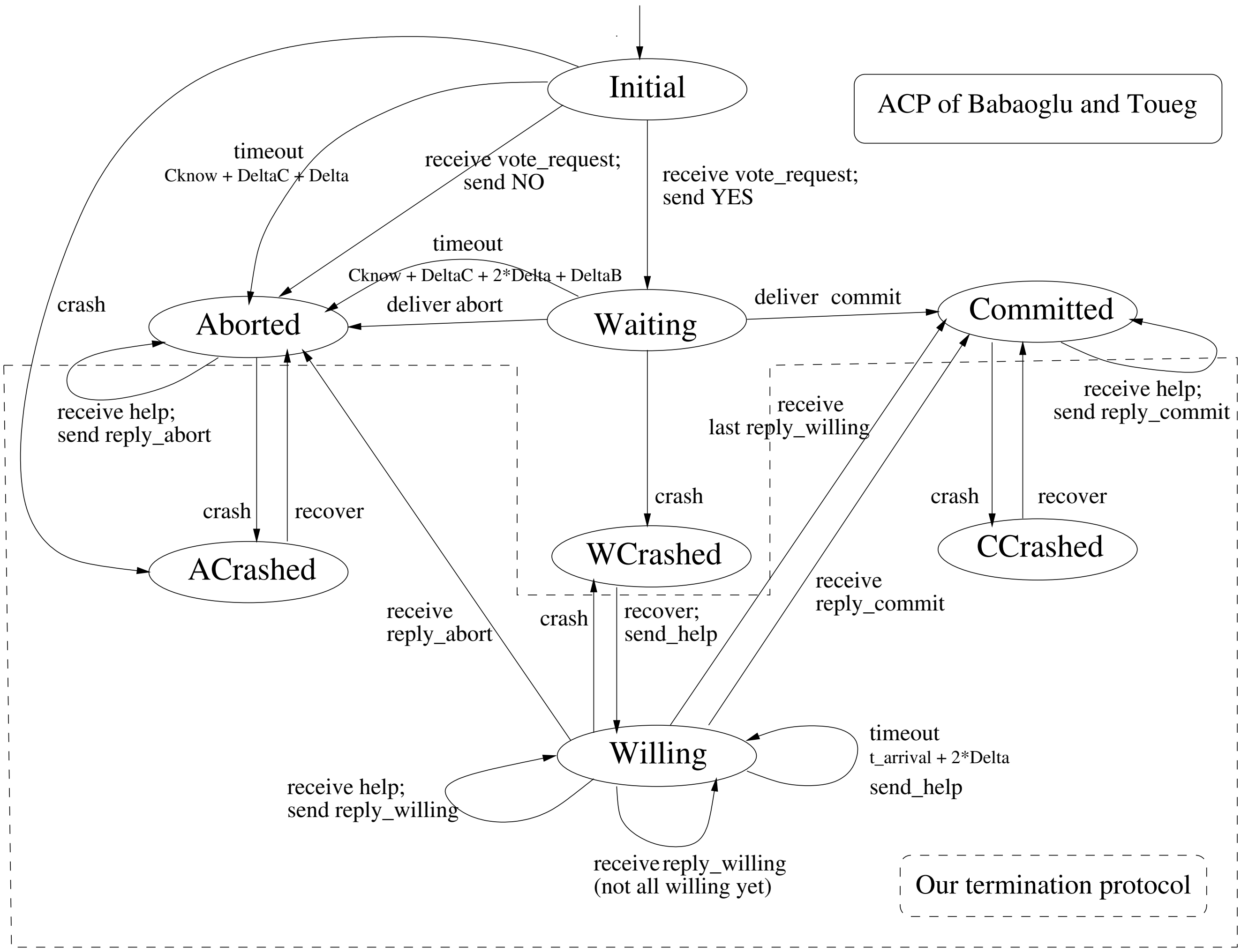

Figure 1. Protocol performed by each participant.

The most obvious way to correct the error is to introduce an additional log record, which would let us distinguish the two states described above (in our notation, `Waiting` and `Willing`).

3. Specification in PVS

We briefly describe the state machines used to specify the protocol, the communication mechanism, and the correctness properties in the language of PVS.

State machines. In general, states are represented in PVS by a record with a number of fields. The main fields are: `name`, to represent the "name" of the state (e.g., `Initial`, `Committed`, etc.); `ident` is the identifier of the corresponding process; `tout` is the time-out of the state (i.e. the time moment before which some action should be taken); `wait` is the set of messages we are waiting for.

In the protocols considered here, there are six actions: send, broadcast, read (representing both receive and deliver), crash, recover and time-out, where the first three have a message as a parameter. Let $time(a)$ denote the execution time of action a.

The protocol performed by a particular process, such as coordinator and participant, is specified by defining the precondition of each action as a predicate on the state and also the effect of each action in terms of a state transition.

Then a run of a process is represented by an infinite sequence of the form $s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \ldots \overset{a_{i-1}}{\to} s_i \overset{a_i}{\to} s_{i+1} \overset{a_{i+1}}{\to} \ldots$, where the s_i are states and the a_i are executed actions, for $i \in \mathbf{N}$. A run should satisfy a number of conditions: each s_i satisfies the precondition of a_i; every pair (s_i, s_{i+1}) corresponds to the effect of a_i; concerning timing, we require $time(a_{i+1}) \geq time(a_i)$, $time(a_i) \leq tout(s_i)$ and non-Zeno behaviour (only finitely many actions in finite time).

In this model, we use infinite runs to represent the behavior of a participant that never completes its termination protocol due to failures of others. Because it is difficult to work in PVS with both finite and infinite runs, we restrict ourself to infinite runs. To implement this, in each of the "final" states (e.g., Committed, ACrashed etc.), where a process

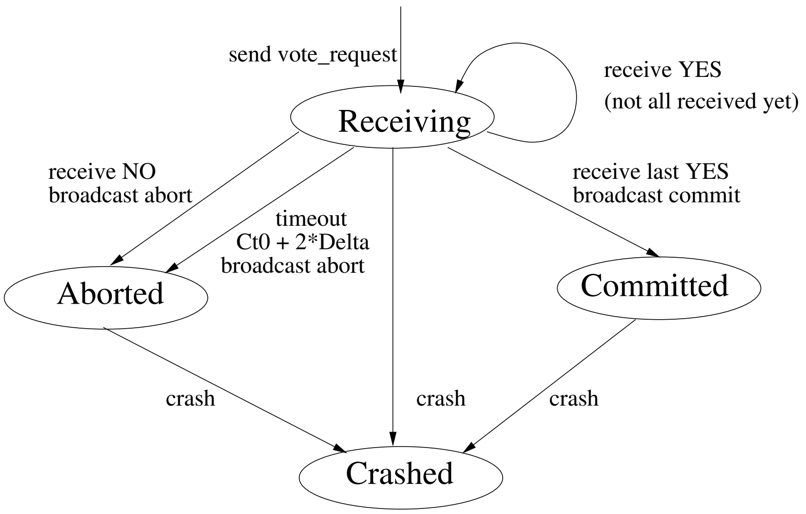

Figure 2. Protocol performed by the coordinator.

may not have to take any action, a fictitious time-out action is introduced, which moves a process to the same state after some fixed unit of time elapses. In this way each finite run is transformed to an infinite run. In an I/O automata framework [Lyn96] both finite and infinite executions of a state machine are considered, but this also leads to a more complex model.

We also associate with each process a record called `Available`, which consists of the set of received messages (plus their receiving time) and the set of delivered messages (plus their delivery time) that are available to the process. Clearly `Available` is related to send and broadcast actions of other processes, as specified by the communication properties mentioned below. The read action is used to get an available message.

A *system execution* is a function which assigns to each process a run and an `Available` record.

Communication mechanism. Since some of the communication properties depend on whether a process has crashed or not, we first introduce some abbreviations in PVS. A process is called `Crashed` between each pair of crash and recover actions and `Correct` otherwise.

Both sending and broadcasting of messages are formalized by a few predicates on system executions. They express properties as validity, integrity, timeliness and uniform agreement. As an example, consider the PVS definition of validity for a broadcast action, where `se` is a system execution. The abbreviation `Broadcast` refers to the corresponding action of the run of `se`, whereas `Deliver` is based on the `Available` record of `se`.

```
BValidity(se) : bool =
 FORALL ps1, ps2, i, ms, t1:
  Broadcast(ps1,ms,t1)(se,i) AND
  NOT Crash(ps1,t1)(se,i+1) AND
  member(ps2, mdest(ms)) AND
  Correct(ps2,t1,t1 + DeltaB)(se)
   IMPLIES
    EXISTS t2:
      t2 > t1 AND t2 < t1 + DeltaB AND
      Deliver(ps2,ms,t2)(se)
```

It expresses that if process `ps1` broadcasts a message `ms` at real-time `t1`, it does not crash during the broadcast (such crash is formalized as a broadcast action immediately followed by a crash action with the same timestamp), some other process `ps2` belongs to the destination of `ms` (expressed by `mdest(ms))` and `ps2` is correct during $[t1,\ t1+\Delta_b]$, then `ps2` delivers `ms` at some moment in this interval.

In our communication model, not all received or delivered messages change the state of a process. A process reacts only to messages which it considers "relevant". This is different from an I/O automata framework [Lyn96], where each state machine is required to be "input enabled".

For instance, suppose that some participant p sends NO vote to the coordinator and moves to the state `Aborted`. After that, the coordinator broadcasts the **abort** decision, which is eventually delivered by all participants including p. The physical delivery of this message is recorded in the `Available` record of p. However, since p has already reached a decision, it will consider this **abort** message irrelevant and will take no action in response to it. Therefore there is no need to specify a logical delivery of this message in the `Aborted` state. Such separation of physical and logical delivery reduces the number of transitions that have to be specified in each state, and therefore simplifies the analysis of a state machine.

If a process is in some state `st`, it only accepts messages that belong to `wait(st)`. The logical delivery is performed by a `read` action. Correctness of a read action is formalized by two predicates: validity and integrity. *Validity* says that if a message `ms` is available at `t1` and a process is waiting for ms, i.e. it is in a state st with ms in wait(st), then this process should read ms at t1. Note that we allow at most one message to be available at any moment of time and a message is lost if it has not been read immediately. *Integrity* expresses that we can read ms at t1 only if ms is available at t1.

Correctness properties. Also the correctness properties are expressed in PVS as predicates on system executions. E.g., using abbreviations Decides and Commits that refer to the occurrence of certain state names in the run, AC1 is represented in PVS as

```
AC1(se) : bool = FORALL p1, p2 :
 (Decides(se)(p1) AND Decides(se)(p2))
  IMPLIES
     (Commits(se)(p1) IFF Commits(se)(p2))
```

Most interesting is the *termination property* AC6. In order to reach a decision, the original formulation requires *all* participants to stay operational simultaneously during some sufficiently long period of time. Our property, however, requires only that *each pair* of participants is operational simultaneously for sufficiently long. This implies termination even for executions in which at most two participants are operational at any moment of time during the execution of the termination protocol. To define this more formally, we introduce the notion of *strongly non-blocking* protocols. We say that an ACP with recovery is strongly non-blocking with respect to a time value T if the following property holds:

AC6S: If a participant p recovers, and after that for each other participant q there is a time interval $[t_q,\ t_q+T]$ such that both p and q stay operational during this interval, then p eventually decides.

We proved that our protocol satisfies AC6S with $T = \Delta_c + 6 * \delta + \Delta_b$, where Δ_c is a given upper bound on electing a coordinator as defined in [BT93a]. T is greater than the time needed to just send a help message and get a reply, because if a participant is still waiting for the coordinator's decision, it will not reply to requests for help from recovered participants. Therefore if some participant recovers "too early", it may have to wait longer.

This improved property is achieved by our termination protocol because of our decision to keep the set of participants that are also willing to commit in stable storage. It is easy to see that if this set is in volatile storage, then there is no T such that the protocol is strongly non-blocking with respect to T. Indeed, suppose that all participants vote YES and then crash while waiting for the coordinator's decision. Next assume that participants p and q recover and stay operational arbitrarily long. Suppose p learns that q is also willing to commit and then they both crash again. If the set of willing participants is in volatile storage, then p would lose the information that q is also willing to commit. If q never recovers, then p will never learn the state of q again and will never decide, even if all participants except q recover and stay operational forever.

4. Verification using PVS

To prove that the protocol satisfies a correctness property, we have to combine operational reasoning about state machines and reasoning in terms of assertions. To illustrate this, we give a sketch of the proof of AC3. Suppose all participants send vote YES to the coordinator and no failures occur. By analyzing the state machine of a participant, we can determine the timing of a YES vote. The validity assertion for the send action implies that the coordinator receives this YES vote before the time-out in state Receiving is triggered.

We also prove that the coordinator cannot leave state Receiving by reading a NO vote from some participant. This follows from the fact that no participant sends a NO vote (which is deduced from the participants state machine) and the integrity of read actions. Hence we prove that the coordinator receives YES votes from all participants before the time-out expires. Since it is waiting for these votes, the validity of read actions implies that the coordinator reads all votes and broadcasts **commit**.

After voting YES, each participant moves into state Waiting. We prove that it cannot read an **abort** decision in this state, since then the integrity assertions for read and broadcast actions would imply that the coordinator broadcasted **abort**. This is impossible, because it broadcasted **commit**. It also cannot crash, because no failures occur. Validity of broadcast implies that a participant delivers the **commit** decision in state Waiting before the time-out expires, and validity of the read action implies that it reads this decision and commits.

Statistics. Verification with the PVS proof checker was performed by the first author and took about four months to complete. He spent one month verifying AC1-AC5 for the basic ACP-BT, then one month verifying the same properties for the complete protocol (in fact only AC1 and AC2, because the proofs of AC3, AC4 and AC5 are almost the same). After that two months were needed to complete the proof of AC6. It is interesting that the assumption about the absence of Zeno behaviour was needed for the last proof.

The table below shows an approximation of the number of interactions with the PVS proof checker that we used to prove each property.

AC1	AC2	AC3	AC4	AC5	AC6	Total
2200	2200	2400	50	150	3000	10000

The proofs of AC4 and AC5 are remarkably easy. However, the proofs of AC1, AC2, AC3 and AC6 are all very complex. The total proof consists of over 250 PVS lemmas and several hours are needed to re-run all the proofs on a UltraSparc workstation. Probably it is possible to simplify the proof and reduce the number of lemmas significantly, but we did not work on this yet. The preliminary version of the PVS specifications and proofs can be found at [URL], and we intend to prepare a more "polished" version as a part of our future work. Note that we simplified some definitions in this paper for the sake of presentation.

5. Concluding remarks

Some work has been done already on the verification of database recovery protocols. E.g., in [LMWF94] a number of very general recovery algorithms for abstract datatypes has been modelled and verified in an I/O automata framework. In [Kuo96], a data manager responsible for single-site recovery has been verified. Simplified versions of 2PC (without timing, site failures and recovery) have been formally treated in [JZ92], using algebraic techniques, and [KST00]. In the last paper a simple safety property has been verified automatically using a games-based model-checking technique. However, we are not aware of any formal verification of more realistic distributed commitment protocols (especially with tool support), perhaps because of their complexity.

We consider our efforts to fill this gap a success. A realistic commitment protocol has been completely verified with an interactive theorem prover. We hope that our specification, based on the combination of state machines and assertions, treats distribution, fault-tolerance and timing in an intuitively acceptable way. It is also close to the informal description of the protocols. Our efforts helped us to find an error in the existing protocol and to clarify ambiguities in its correctness properties.

We conclude that completely formal or even mechanical verification of ACP's is feasible now. In general, we advocate a modular approach to specification and verification, which abstracts from the less important details of message processing and log management. This would lead to specifications that are high-level enough and therefore make the complexity manageable.

The authors have also done some work on verification of concurrency control protocols [CHvdS99] and in future research we hope to combine these protocols with single-site recovery and distributed commitment protocols, in order to prove atomicity and serializability for a complete transaction processing system.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Comp., 1987.

[BT93a] O. Babaoglu and S. Toueg. Non-blocking atomic commitment. In S. Mullender, editor, *Distributed Systems*, pages 147–168. Addison-Wesley Publishing Comp., 1993.

[BT93b] O. Babaoglu and S. Toueg. Understanding non-blocking atomic commitment. Technical Report UBLCS-93-2, University of Bologna, 31 pages, ftp://ftp.cs.unibo.it/pub/ozalp/Misc/NB-AtomicCommit.ps.gz, 1993.

[CHvdS99] D. Chkliaev, J. Hooman, and P. van der Stok. Serializability preserving extensions of concurrency control protocols. In *Proc. of the A. Ershov Third Int'l Conf. Perspectives of System Informatics*, pages 179–192. LNCS 1755, 1999.

[Hoo97] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Proc. AMAST Conference*, pages 261–275. LNCS 1349, 1997.

[JZ92] W. Janssen and J. Zwiers. Protocol design by layered decomposition: A compositional approach. In *Proc. FTRTFTS 2nd Int'l Symposium*, pages 307–326. LNCS 571, 1992.

[KST00] T. Kempster, C. Stirling, and P. Thanisch. Games-based model checking of protocols: counting doesn't count. In *Workshop on Distributed System Validation and Verification (DSVV'2000),* This volume, 2000.

[Kuo96] D. Kuo. Model and verification of a data manager based on ARIES. *ACM Transactions on Database Systems*, 21(4):427–479, 1996.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., 1994.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[PVS] *Prototype Verification System,* For more details, see http://pvs.csl.sri.com/.

[URL] *PVS specifications and proofs,* See http://www.cs.kun.nl/∼hooman/ACP.html.

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