Mechanical Verification of a Non-Blocking Atomic Commitment Protocol

Dmitri Chkliaev

Peter van der Stok

Eindhoven University of Technology

Jozef Hooman

University of Nijmegen

Non-blocking atomic commitment protocol of Babaoğlu and Toueg verified using interactive theorem prover **PVS**.

PVS model close to informal text:

- Pseudo code of distributed processes formalized by *timed state machines*; interleaving of actions leads to (global) execution sequences.
- Informal properties of correctness and communication between processes (e.g. broadcast) formalized by *predicates on execution sequences*. Leads to improved formulation.

Verification system **PVS**

developed by SRI, California.

Specification language of PVS: typed higher-order logic with records, abstract datatypes, built-in types as reals, sequences, etc.
Structuring by parameterized theories.

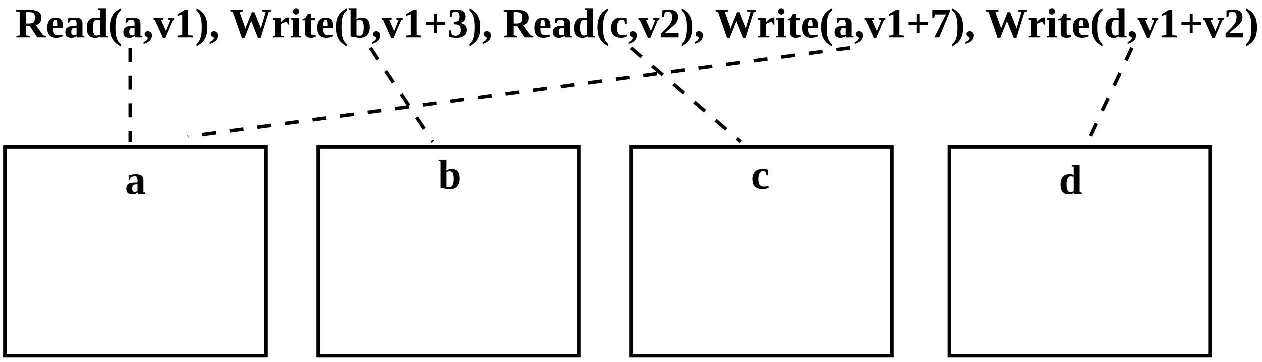
PVS system contains interactive theorem prover with powerful decision procedures.

Here interactive verification of correctness using PVS combines reasoning on state machines and assertions.

Error found in termination protocol (to deal with recovery of crashed sites) of Babaoğlu and Toueg.
Own improved protocol mechanically verified.

Context:

Distributed database:
transactions contain read and write actions
on variables at different sites.



Actions at sites might fail
(due to memory failure, conflict, deadlock, ..).

Requires **atomicity**: all updates or none.

Achieved by
atomic commitment protocol (ACP):

all commit or all abort.

Sites communicate by exchanging messages.

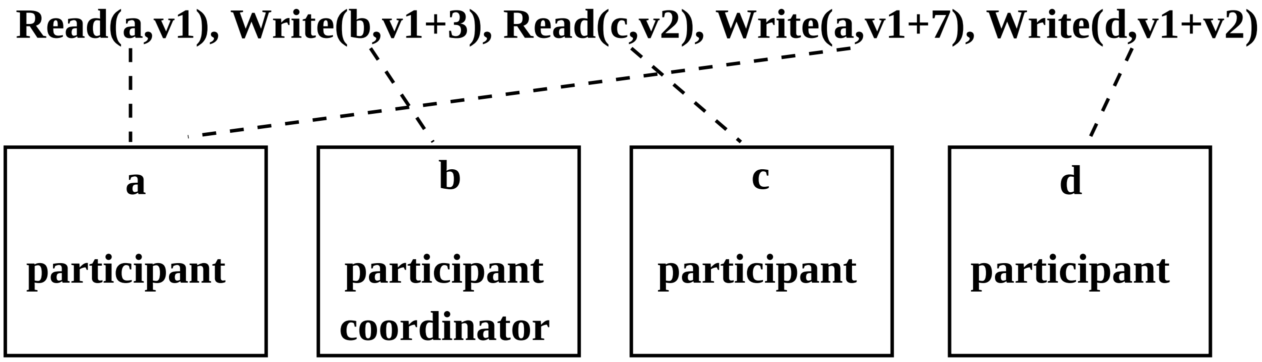
There are known bounds on message delays.

Most popular ACP:

the <u>**Two-Phase Commit**</u> (2PC) protocol.

Participant protocol at each site and

one **coordinator** (elected within Δ_c).



<u>**Pseudo code Coordinator**</u>

send *vote_request* **to all** participants

set-timeout-to *local_clock* $+ 2\delta$

wait-for (receipt of *vote* messages
from all participants)

  if (all votes are *YES*)

  then broadcast(COMMIT, participants)

  else broadcast(ABORT, participants)

on-timeout broadcast(ABORT, participants)

Pseudo code Participants

Without recovery from failures.

At time C_{know} participants know

about transaction.

set-timeout-to $C_{know} + \Delta_c + \delta$

wait-for (receipt of *vote_request* from
coordinator)

  send *vote* **to** coordinator

  if (*vote* = *NO*) **then** decide ABORT

  else

    set-timeout-to $C_{know} + \Delta_c + 2\delta + \Delta_b$

    wait-for (delivery of decision message)

      if (decision message is ABORT)

      then decide ABORT

      else decide COMMIT

    on-timeout decide ABORT

on-timeout decide ABORT

Properties Communication Mechanisms

For **send & receive**:

"each message is received within δ time units after being sent".

Simple Broadcast

B1 (Validity):

If correct process broadcasts m

then all correct processes eventually deliver m.

B2 (Integrity):

Each process delivers m at most once and only if some process broadcasts m.

B3 (Δ_b-Timeliness):

If broadcast of m initiated at t,

no process delivers m after $t + \Delta_b$.

Note: $\Delta_b \geq \delta$.

Correctness properties

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).

With Simple Broadcast AC1 - AC4 hold.

Additional requirement:
all correct participants terminate, i.e.
<u>*non-blocking* ACP</u>:

AC5: Every *correct* participant,
i.e. one that never crashes,
eventually decides.

2PC might <u>block</u> when using
simple, *unreliable*, broadcast;
e.g. when coordinator crashes during broadcast
of commit and only faulty participants deliver;
if they do not recover, no correct participant
can decide.

But this 2PC is *non-blocking* when using
<u>*Reliable Broadcast*</u>:
B1, **B2**, **B3**, and

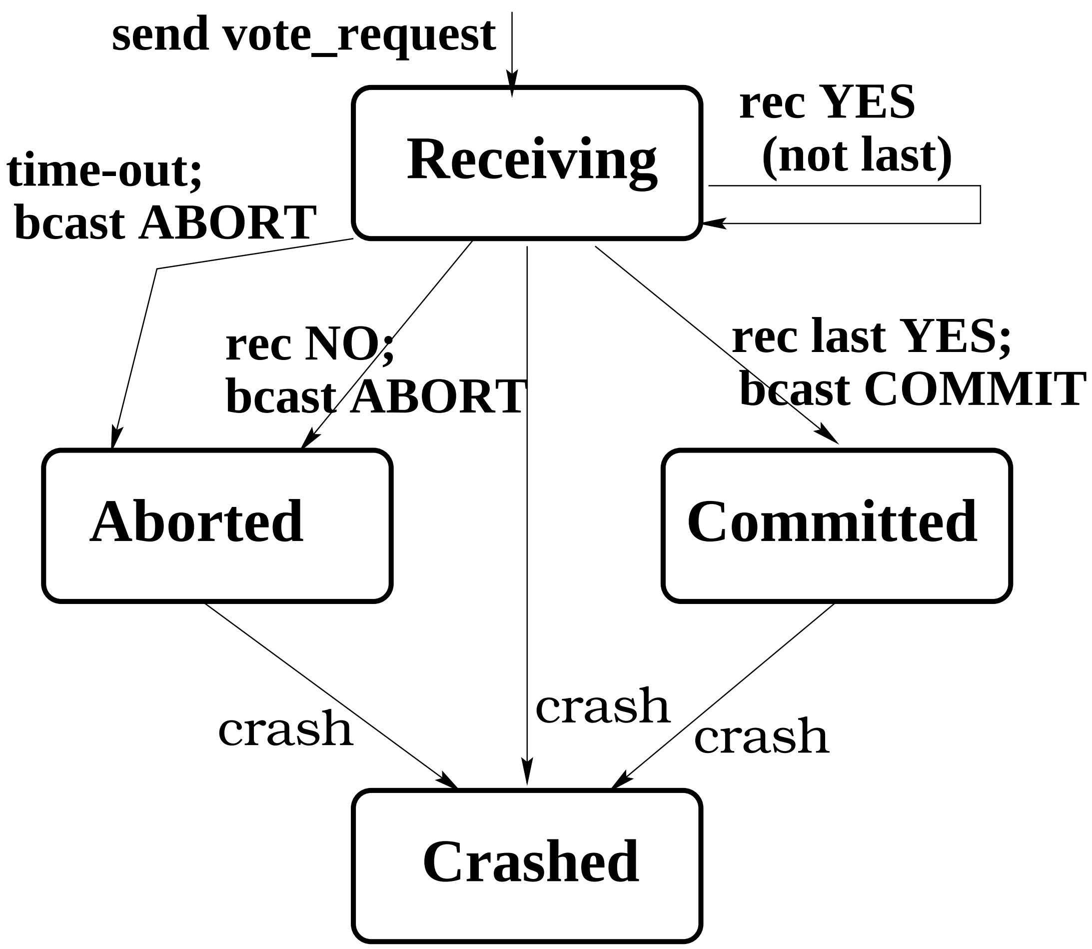
<u>**B4** (Uniform Agreement):</u>
If any process (correct or not) delivers m,
then all correct processes eventually deliver m.

Basic 2PC with Reliable Broadcast

Modeled and verified using PVS.

PVS model close to informal formulation: protocol of coordinator and participants represented by timed state machines.

Coordinator



States, denoted s, s_0, s_1, ... in PVS:

```
States : TYPE =
  [# name  : StateNames,
     ident : Processes,
     tout  : Time,
     wait  : pred[Messages],
     ...   : ... #]
```

Six actions, denoted a, a_0, a_1, ..:
send, **broadcast**,
read (representing both receive and deliver),
crash, **recover** (used later) and
time-out.

Actions may have parameters (e.g. message, process id) and contain a time stamp, $time(a)$. One transaction, no transaction identifiers.

Protocol specified by defining

- precondition of each action as a predicate on the state and
- effect of executing each action as state transition, i.e. predicate on pair of states.

Run of process represented by infinite sequence:

$s_0 \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \ldots \stackrel{a_{i-1}}{\rightarrow} s_i \stackrel{a_i}{\rightarrow} s_{i+1} \stackrel{a_{i+1}}{\rightarrow} \ldots$

satisfying:

- s_i satisfies precondition of a_i;
- (s_i, s_{i+1}) corresponds to effect of a_i;
- $time(a_{i+1}) \geq time(a_i)$;
- $time(a_i) \leq tout(s_i)$;
- non-Zeno behaviour.

System execution: assigns to each process a run and an `Available` record, denoting set of received and delivered messages at any time.

```
Available : TYPE =
  [# receive : [[Messages, Time] -> bool],
     deliver : [[Messages, Time] -> bool] #]
```

Read actions used to get available message.

Communication mechanism.

Send/receive properties, defined in PVS as predicates on system executions, following informal text plus additional properties (e.g. integrity; message received has been sent).

Similarly, define **properties of broadcast.**

Let `se` be a system execution.

Process is `Crashed` between crash and recover actions and `Correct` otherwise.

`Deliver` is based on `Available` record of `se`.

Property **B1**:

```
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)
```

In property **B4** of reliable broadcast,
the broadcasting process has been introduced
to make correctness period precise.

```
BUnifAgreement(se) : bool =
 FORALL ps1, ps2, i, ms1, t1 :
  Broadcast(ps1,ms1,t1)(se,i) AND
  (EXISTS ps3, t3 : Deliver(ps3,ms1,t3)(se)) AND
  mdest(ms1)(ps2) AND
  Correct(ps2,t1,t1 + DeltaB)(se)
    IMPLIES
     EXISTS t2 : t2 > t1 AND
                 t2 < t1 + DeltaB AND
                 Deliver(ps2,ms1,t2)(se)
```

These predicates reduce the set of possible
system executions.
Used instead of axioms to make dependencies
explicit in lemmas.

Correctness properties.

Also expressed in PVS as predicates on system executions.

Abbreviations `Decides` and `Commits` refer to occurrence of state names in the run.

```
Decides(se)(p1) : bool =
  Commits(se)(p1) OR Aborts(se)(p1)

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

AC5(se) : bool = FORALL p1 :
   Correct_always(se, p1)
      IMPLIES Decides(se)(p1)
```

Correctness proof:

Constructed interactively with PVS proof checker.

Reasoning about transitions in state machines, relating processes by communication predicates.

Recovery from Failures

Suppose participant can recover after failure.

Recovering participant returns to operational state using termination protocol;

First restore state using stable storage.

Next try to finish running transactions.

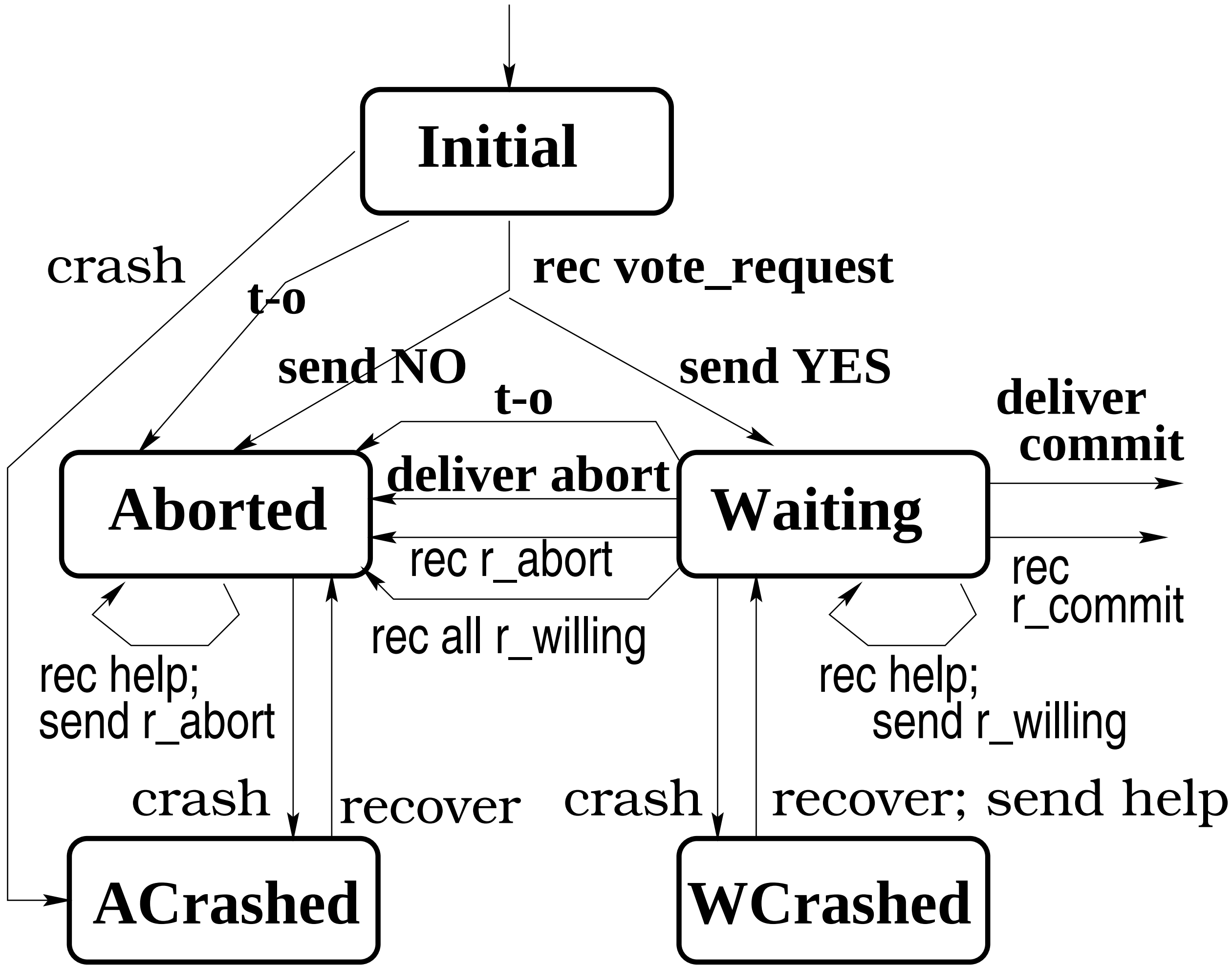
Required *termination property*:

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

We improved this formulation;

first consider termination protocol of Babaoğlu and Toueg.

Protocol Participant (omitting commit)



Error:

All vote YES

Coordinator broadcasts COMMIT

p crashes, p recovers

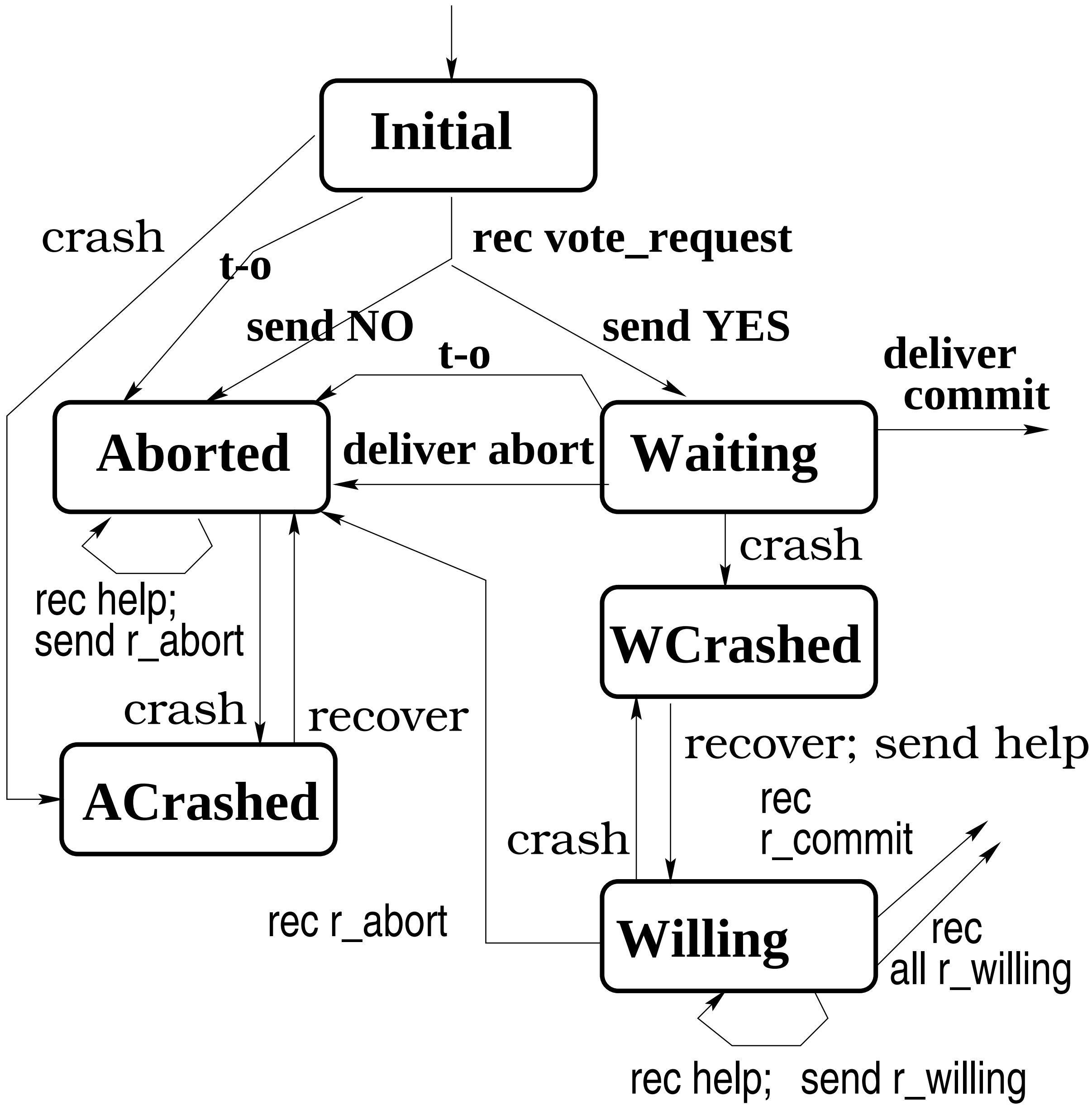
p gets reply_willing from all others

p decides ABORT

Others deliver & decide COMMIT

Improved Protocol Participant

Distinguishing Waiting and Willing; decide COMMIT if all willing (i.e. have voted YES).



Spec. and Verification in PVS

Improvement of *termination property*:

Informally

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.

Correctness proof:

Proofs of AC1 - AC5 re-used and modified
(mainly for AC1 and AC2).

Protocol satisfies **AC6S** with

$$T = \Delta_c + 6\delta + \Delta_b$$

(proof of AC6S uses non-Zenoness).

Concluding Remarks

Only *properties* of broadcast mechanism used,
implementation is separate issue;
has been done before (with Ping Zhou).

Other protocols verified with **PVS**:
parts of ACCESS.bus and IEEE P1394,
and membership protocol.

Related to work presented here:
serializability of *concurrency control protocols*
(two-phase locking, timestamp ordering).
Plus correctness preserving extensions.
Also for ACP we would like to formalize
our incremental verification.

Current/future work also includes
verification of ACID properties of
more general *transaction processing systems*,
integrating strict two-phase locking,
undo/redo recovery and two-phase commit.

For more info and PVS dump file, see
`http://www.cs.kun.nl/~hooman`