Model Checking a Modular-Structured Nonblocking Atomic Commitment Protocol for Asynchronous Distributed Systems

Eun-Hye CHOI [1]
Tatsuhiro TSUCHIYA [2] Tohru KIKUNO [2]

1: AIST CVS 2: Osaka University

Model Checking a Modular-Structured Nonblocking Atomic Commitment Protocol for Asynchronous Distributed Systems

Eun-Hye CHOI[†] Tatsuhiro TSUCHIYA[‡] Tohru KIKUNO[‡]
[†]Research Center for Verification and Semantics,
National Institute of Advanced Industrial Science and Technology (AIST),
5th Floor, 1–2–14 Shin-Senri Nishi, Osaka 560–0083 Japan
[‡]Graduate School of Information Science and Technology, Osaka University
1–5 Yamadaoka, Suita, Osaka 565–0871, Japan
[†] e.choi@aist.go.jp [‡]{t-tutiya, kikuno}@ist.osaka-u.ac.jp

Abstract

Various principal fault-tolerant agreement protocols for asynchronous distributed systems, such as atomic commitment and view synchrony, can be constructed in a modular way which is based on consensus and failure detectors. Since, however, it is difficult to design correct fault-tolerant distributed protocols especially for asynchronous systems, the development of an efficient framework for verifying the fault-tolerant distributed agreement protocols is of importance. In this paper, we focus on a modular-structured nonblocking atomic commitment (NBAC) protocol as a case study and propose a method to verify the protocol by model checking. In the proposed method, we first construct a model for the NBAC protocol in a modular way by composing a behavior model for unreliable failure detector and a behavior model for distributed computing nodes participating in the target protocol. We next construct temporal logic formulae expressing the termination, justification, and obligation properties of the NBAC protocol assuming that the properties that the consensus module and the unreliable failure detectors should provide are guaranteed. Finally, the efficiency of our method is evaluated through the experimental results obtained from using two model checking tools, SPIN and SMV. We conjecture that our assume-guarantee model checking approach given in this paper is applicable to generic modular-structured fault-tolerant agreement protocols for asynchronous distributed systems.

1 Introduction

Design and verification of fault-tolerant distributed protocols are complex and difficult especially for asynchronous distributed systems because of unbounded

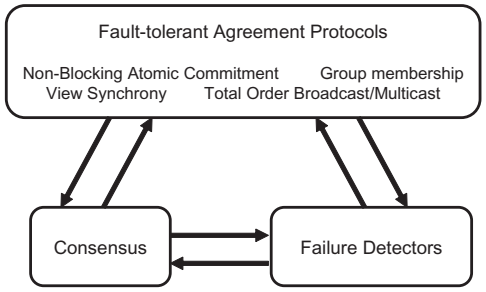

Figure 1: Modular Architecture of Distributed Agreement Protocols.

communication delays and process relative speeds. For example, it is known that designing a deterministic algorithm to solve *consensus*, which is the most fundamental and important problem of distributed systems, is impossible for asynchronous distributed systems even with a single crash failure [4].

In order to handle this problem, the concept of *unreliable failure detectors* has been introduced [2]. Unreliable failure detectors are equipments that make the consensus problem solvable in asynchronous distributed systems. Based on the consensus algorithm and unreliable failure detectors, various principal fault-tolerant agreement protocols, such as *nonblocking atomic commitment*, *group membership*, *view synchrony*, and *total order broadcast/multicast*, can be constructed in a modular way [6].

Figure 1 shows the generic modular architecture of the fault-tolerant distributed agreement protocols. Consensus is used as a black-box module and failure detectors are used not only in the consensus module but also outside of the consensus module. The properties that failure detectors are required to satisfy depend on the target distributed agreement protocol.

The aim of our research is to develop an efficient framework for automatically verifying the generic modular-structured fault-tolerant distributed protocols based on consensus and unreliable failure detectors. In this paper, we focus on the modular-structured nonblocking atomic commitment (NBAC) protocol [5, 11] as a case study and propose a method for verifying the *termination*, *justification*, and *obligation* properties of the NBAC protocol by *model checking*.

Model checking [3, 7, 9] is a verification technique that exhaustively checks whether or not a given finite state transition system satisfies a given temporal logic property. Since, given a transition system and a temporal logic formula, model checking can be automatically and rapidly performed using existing tools like SPIN [7] and SMV [9], it has recently regained the attention of researchers. Model checking is also helpful in locating system errors, since whenever a system model fails to satisfy a property formula, a counterexample that shows the system behavior that violates the property is produced as the output.

In the proposed method, we first construct a model for the NBAC protocol in a modular way by composing asynchronous processes representing unreliable failure detectors and distributed computing nodes participating in the target

protocol. We next construct temporal logic formulae expressing the termination, justification, and obligation properties for the model assuming the guaranteed properties by the consensus module and the unreliable failure detectors. Our *assume-guarantee* model checking approach is useful to improve a verification speed and to reduce state space needed for verification. Finally, we present the experimental results of verifying if the constructed model satisfies the constructed properties by using the two existing model checking tools, SPIN and SMV.

2 Preliminaries

2.1 System Model

We consider an *asynchronous* distributed system consisting of a set of computing nodes, $\mathcal{P} = \{p_0, \cdots, p_{N-1}\}$, completely connected by communication links. All computing nodes are autonomous, have no shared memory, and communicate with each other by reliable message passing. Each computing node has two states: operational or failed (*crash-stop*). There is no bound on communication delays and process relative speeds.

2.2 Consensus

Consensus [4] is a fundamental problem of distributed systems. Each computing node has an input value and all operational computing nodes have to decide on a common output value. The consensus problem comprises the following three properties:

- **Termination**: Every operational node eventually decides some value.
- **Agreement**: No two operational nodes decide different values.
- **Validity**: If a node decides a value, then the value is the input value of some node.

The *uniform consensus* problem is defined by replacing the agreement property with the following stronger requirement:

- **Uniform Agreement**: No two nodes decide different values.

The consensus and uniform consensus problems have relatively simple solutions for synchronous distributed systems, while it is impossible to design a deterministic protocol solving consensus for asynchronous distributed systems even with a single crash failure [4]. Because of this impossibility, researchers have investigated a set of minimal properties, like *unreliable failure detectors* [2], that make the asynchronous consensus problem solvable.

Table 1: Failure Detector Classes

Completeness	Accuracy			
	Strong	Weak	◇Strong	◇Weak
Strong	$\mathcal{P}$	$\mathcal{S}$	$\diamond\mathcal{P}$	$\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	$\mathcal{W}$	$\diamond\mathcal{Q}$	$\diamond\mathcal{W}$

2.3 Failure Detectors

Since communication delays and process relative speeds are not bounded in asynchronous systems, it is impossible to distinguish a node whose message or speed are very slow from a failed node. Therefore, failure detection with *completeness* and *accuracy* is impossible in asynchronous systems. Completeness means that a failed node will eventually be detected, while accuracy means that an operational node will not be considered failed.

The concept of *unreliable failure detectors* [2] has been introduced to solve the consensus problem in asynchronous systems. A failure detector is a distributed oracle which gives hints on failed nodes. Each failure detector D_i for computing node p_i maintains a set of nodes that it currently *suspects* to have failed, and each node p_i has access to failure detector D_i to get the hints on failed nodes.

Failure detectors are *unreliable* because they can suspect an operational node or not suspect a failed node. Failure detectors are classified to the eight classes in Table 1 with the following two completeness properties and four accuracy properties:

- **Strong Completeness**: Eventually every failed node is permanently suspected by *every* operational node.
- **Weak Completeness**: Eventually every failed node is permanently suspected by *some* operational node.
- **Strong Accuracy**: *Every* operational node is never suspected before it failed.
- **Weak Accuracy**: *Some* operational node is never suspected.
- **Eventual Strong Accuracy**: *Eventually every* operational node is never suspected by any operational node.
- **Weak Accuracy**: *Eventually some* operational node is never suspected by any operational node.

In [2], Chandra and Toueg stated that unreliable failure detectors satisfying only weak completeness and eventual weak accuracy make the asynchronous consensus problem solvable.

3 Nonblocking Atomic Commitment (NBAC)

In a distributed system, a transaction involves several computing nodes as data manager processes. Just before the completion of a transaction, nodes have to solve an *atomic commitment* problem [1] in order to decide on *commit* or *abort* of the transaction. If the decision is *commit*, all nodes make their updates permanent; if the decision is *abort*, all their temporary writes are ignored.

The decision depends on votes from the nodes. Each node votes *yes* if for example no concurrency control conflict has been detected locally; otherwise the node votes *no*. Atomic commitment requires that *commit* must be decided if all votes are *yes* and no node has failed.

Nonblocking atomic commitment [12] requires that an operational node must make a decision even if some nodes have failed. The nonblocking atomic commitment problem is specified by the following four properties:

- **Termination**: Every operational node eventually decides.
- **Uniform Agreement**: No two nodes decide differently.
- **Justification**: If a node decides *commit*, all nodes have voted *yes*.
- **Obligation**: If all nodes have voted *yes* and there is no failure, then the decision is *commit*.

Termination is a liveness property which guarantees nonblocking. *Uniform agreement* and *justification* are safety properties. *Obligation* is a liveness property to eliminate a trivial solution that always output *abort*. Since failures can only be suspected erroneously in asynchronous systems, the *obligation* property is weakened for the problem to be solvable as follows:

- **Obligation** (***in asynchronous systems***): If all nodes have voted *yes* and there is no *failure suspicion*, then the decision is *commit*.

Figure 2 describes the generic modular-structured NBAC protocol in asynchronous systems. The protocol is constructed based on a uniform consensus module, *UniformConsensus*(i) with an input value i, and unreliable failure detectors, $\mathcal{D} = \{D_0, \cdots, D_{N-1}\}$, as follows: First, each node sends its vote to all nodes. Next, each node waits until (a) it receives a vote *no* from some node, (b) its failure detector suspects some node, or (c) it receives vote *yes* from all nodes. Finally, each node calls a uniform consensus module by proposing *abort* in the cases of 2.(a) and 2.(b) and by proposing *commit* in the case of 2.(c), and decides the outcome returned by the uniform consensus module.

In order to resolve NBAC, the protocol in Figure 2 requires the failure detectors to satisfy the strong completeness property [5, 11].

4 Model Checking

Model checking [3, 7, 9] is a verification technique that exhaustively checks whether or not a system modeled as a finite state transition system satisfies

For each node p_i,
1. Send its vote $v_i \in \{yes, no\}$ to all nodes.
2. Wait until
   (a) it receives a vote *no* from some node,
   (b) $\exists j : p_j \in S_i$ (where S_i is the set of nodes suspected by D_i), or
   (c) it receives vote *yes* from all nodes.
3. Call the uniform consensus module and decide the outcome:
   3.1) 2.(a) and 2.(b) $\rightarrow$ outcome = *UniformConsensus*(*abort*)
   3.2) 2.(c) $\rightarrow$ outcome = *UniformConsensus*(*commit*)

Figure 2: Modular-structured NBAC Protocol based on Consensus [5, 11].

the property expressed by the temporal logic formula. Since, given a transition system and a temporal logic formula, model checking can be automatically and rapidly performed using existing tools, it has recently regained the attention of researchers. Model checking is also helpful in locating system errors, since whenever a system model does not satisfy a property formula, a counterexample that gives the system behavior that violates the property is given as the output. Many model checking tools have been developed so far, and among them, SPIN [7] and SMV [9] are the best known and most powerful ones.

An input model to SPIN is described in *Promela* (*Process Meta-Language*) which has C-like syntax. A Promela model consists of one or more asynchronous processes with data objects, non-deterministic constructs, and communication primitives. Processes can communicate via synchronous and asynchronous message passing with buffered channels or shared memory. SPIN verifies the claims specified by the Linear Temporal Logic (LTL) [8] formulae or process invariants, which can express the basic safety and liveness properties. SPIN performs on-the-fly verification and supports several useful state search and compression strategies.

An input model to SMV(*Symbolic Model Verifier*) is described in SMV language. An SMV model consists of one or more modules specifying finite state machines which are specified as either synchronous or asynchronous processes. Processes can interact with each other using shared variables but not provided communication channels, differently from Promela. SMV basically verifies the Computation Tree Logic (CTL) [8] formulae. It is known that SMV can handle relatively larger state space compared to SPIN because of the use of compact symbolic representations of the state space.

With our research, we used both model checking tools SPIN and SMV for verifying the target NBAC protocol. We used SPIN because asynchronous distributed computing nodes and failure detectors participating in the target protocol can naturally be modeled as asynchronous processes communicating via channels in Promela. We also used SMV for comparison purposes.

5 Modeling of NBAC Protocol

In this section, we first explain the modeling of the target NBAC protocol in Promela in Section 5.1. We next specify LTL formulae expressing the the NBAC properties for our model in Section 5.2. In Section 5.3, we also present the modeling in SMV and specifying CTL properties for the target NBAC protocol.

5.1 Modeling in Promela

Figure 3 shows the Promela model for the NBAC protocol where the number of nodes, N, is 2. The model can be straightforwardly extended to $N > 2$.

The model declares two types of processes named `node` and `detector` using the `proctype` keyword. The process `node` represents a computing node participating in the NBAC protocol. The process `detector` represents a failure detector for each node. In the `init` section (Lines 23–30), N processes of `node` and N processes of `detector` are simultaneously instantiated using the `run` keyword. In the Promela execution semantics, a sequence of statements in an `atomic`-statement are executed indivisibly. Promela processes are executed concurrently and scheduled nondeterministically.

In the following, we first explain the modeling of the failure detectors in Section 5.1.1 and next present the modeling of the nodes in Section 5.1.2.

5.1.1 Modeling of Unreliable Failure Detectors

The behavior of an unreliable failure detector is declared in Lines 32–47 of Figure 3. Process `detector(i)` models the behavior of each failure detector D_i such that it arbitrarily determines for each of the other nodes whether it is suspected or not.

In the model, $S[i]$ denotes the set of nodes that failure detector D_i currently suspects and $S[i].el[j] = 1$ represents that $p_j \in S[i]$. In addition, $ex\_sus[i] = 1$ means that there is a failure suspicion by failure detector D_i. In the Promela execution semantics, each statement is either blocked or executable. As for a `do`-statement and an `if`-statement, if more than one statement are executable, that is, the guards of the statements are true, one of the statements is nondeterministically selected and executed. Therefore, by Lines 32–40 and Lines 43–46, the value of $S[i].el[j]$ is determined arbitrarily as 0 or 1.

As mentioned in Section 3, the target NBAC protocol assumes the strong completeness of the failure detectors used in the protocol. For verification, we represent the assumed strong completeness of the failure detectors by a temporal logic formula and the formula will be in turn used as the required properties for the model of the failure detectors as will be explained in Section 5.2.

The strong completeness of the failure detectors is the property that eventually every failed node is permanently suspected by every operational node. This property for the model in Figure 3 is expressed in LTL as follows:

$$\Box(\exists p_i \in \mathcal{P}.(crash[i]) \rightarrow \Diamond\Box(\forall p_j \in \mathcal{P}.(crash[j] \vee S[j].el[i])))$$

```
#define N 2 /* # of nodes */

bool crash[N];
 /* crash[i]=1 : node p_i crashed */
typedef Array{
  bool el[N]
};
Array S[N];
 /* set of nodes suspected by failure detector D_i */
bool ex_sus[N];
 /* ex_sus[i]=1 : there is a failure suspicion by D_i*/
bool vote[N];
 /* vote[i]=1 or 0: vote of node p_i is Yes or No */
bool propose[N];
 /* propose[i]=1 or 0 :
    proposition of node p_i is Commit or Abort */
bool fin[N];
 /* fin[i]=1 :
    p_i has crashed or called the consensus module*/
chan ch2p[N] = [N] of {bool};
 /* ch2p[i] : message channel to node p_i*/

init{
  atomic{
    run node(0);
    run node(1);
    run detector(0);
    run detector(1);
  }
}

inline guess(j){
  /* determine whether D_i suspects p_j or not */
  atomic{
    if
      :: true -> S[i].el[j] = 0
      :: true -> S[i].el[j] = 1; ex_sus[i] = 1
    fi
  }
}

proctype detector(byte i){
  do
    :: true -> guess(0)
    :: true -> guess(1)
  od
}

inline voting(){
  /* decide a vote */
  if
    :: true -> vote[i] = 1;
    :: true -> vote[i] = 0
  fi
}

inline sending(){
  byte j;

  /* send the vote to all */
  do
    :: j<N -> ch2p[i]!vote[i]; j++
    :: j>=N -> break
  od
}

proctype node(byte i) {
  byte yesnum; /* current # of received Yes */
  bool flag;

  voting(); flag=1;
  do
    :: flag -> sending(); flag=0
    :: !flag && ch2p[i]?[1] && !ex_sus[i]
       -> ch2p[i]?_; yesnum++
    :: !flag && ch2p[i]?[0] /* cond 2.(a) */
       -> ch2p[i]?_; propose[i] = 0; break
    :: !flag && ex_sus[i] /* cond 2.(b) */
       -> propose[i] = 0; break
    :: !flag && yesnum==N /* cond 2.(c) */
       -> propose[i] = 1; break
    :: true /* crash-stop failure */
       -> crash[i] = 1; break
  od;
  fin[i]=1
}
```

Figure 3: Promela Model for the NBAC Protocol (with N=2).

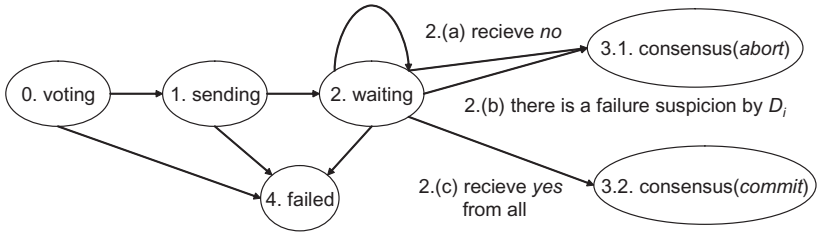

Figure 4: Overview of a Node Behavior in the NBAC Model.

where $crash[i]$ represents that node i has failed (crash-stopped).

5.1.2 Modeling of NBAC Nodes

The behavior of node p_i in the NBAC protocol is declared as Process `node(i)` in Lines 49–86 of Figure 3.

Figure 4 shows the overview of the behavior of node p_i as a finite state machine. Each node p_i starts from State 0 (the *voting* state) where p_i nondeterministically decides its vote (*yes* or *no*). State 1 (the *sending* state) corresponds to Step 1 of the protocol in Figure 2. In State 1, the node sends its vote to all nodes. This is described in Lines 49–65 in the Promela model. In the model, message passing to node p_i is performed using buffered channel `ch2p[i]`.

Thereafter, if node p_i is operational, the state of node p_i transits to State 2 (the *waiting* state) which corresponds to Step 2 of the protocol. The transition from State 2 to State 3.1 (3.2) corresponds to Step 3.1 (3.2) in Figure 2 and is described in Lines 76–79 (Lines 80–81) of the Promela model.

In the model, Lines 74–75 describes the self-loop transition of the *waiting* state, and Lines 82–83 describes the transitions from States 1 and 2 to State 4 which correspond to the crash-stop failure of the node. In States 3.1 and 3.2 in Figure 4, the uniform consensus module is called by proposing *abort* and *commit*, respectively.

To carry out the verification of the NBAC protocol, we use an assume-guarantee method where instead of modeling the behavior of the uniform consensus algorithm, the properties it should provide are only taken into consideration.

5.2 Specifying Required LTL Properties

NBAC requires the four properties, *Termination*, *Uniform Agreement*, *Justification*, and *Obligation*, as mentioned in Section 3. Among the four properties, the *uniform agreement* property obviously holds for the target NBAC protocol, because the target protocol determines the outcome using the uniform consensus module which satisfies the uniform agreement property. We thus focus on the remaining three properties: the *termination* property, the *justification* property, and the *obligation* property.

The target protocol is constructed using the two modules: the uniform consensus and the unreliable failure detectors with the strong completeness. The target protocol is hence required to satisfy the NBAC properties, provided that (A1) the guaranteed properties by the uniform consensus and (A2) the strong completeness of the unreliable failure detectors are satisfied.

We first explain the required properties for the proposed model of the target protocol in Figure 3 assuming only (A1), the guaranteed properties of the uniform consensus module.

First, consider the termination property, which is the property that every operational node eventually decides. The termination is guaranteed for the uniform consensus module. Hence, the termination property for the proposed model using the uniform consensus module is equivalent to the property that every operational node eventually calls the uniform consensus module. This property is expressed in LTL as follows:

$$P_t = \Diamond(\forall p_i \in \mathcal{P}.(fin[i]))$$

where $fin[i]$ means that node i calls the uniform consensus module if the node is operational. (In the proposed model, $fin[i] = 1$ if and only if each node i reached the states where node i calls the uniform consensus module or node i fails.)

Next, consider the justification property, which is the property that if a node decides *commit*, all nodes have voted *yes*. By the validity of the uniform consensus module, if a node decides *commit*, some node has proposed *commit*. Therefore, the justification property for the proposed model using the uniform consensus module is equivalent to the following property: If a node proposes *commit*, all nodes have voted *yes*. This property is expressed in LTL as follows:

$$P_j = \Box(\exists p_i \in \mathcal{P}.(propose[i] = commit) \rightarrow \forall p_j \in \mathcal{P}.(vote[j] = yes))$$

where $propose[i]$ and $vote[i]$ respectively denote the value proposed to the uniform consensus module and the vote for each node i.

Thirdly, consider the obligation property, which is the property that if all nodes have voted *yes* and there is no failure suspicion, then the decision is *commit*. The contraposition of this property is that if the decision is not *commit*, that is, the decision is *abort*, then some node has not voted *yes* or there is a failure suspicion. By the validity of the uniform consensus module, if a node decides *abort*, some node has proposed *abort*. Therefore, the obligation property for the proposed model using the uniform consensus module is equivalent to the following property: If a node proposes *abort*, some node has not voted *yes* or there is a failure suspicion. This property is expressed in LTL as follows:

$$P_o = \Box(\exists p_i \in \mathcal{P}.(propose[i] = abort) \rightarrow \exists p_j \in \mathcal{P}.(vote[j] \neq yes \vee ex\_sus[j]))$$

where $ex\_sus[i]$ means that there is a failure suspicion by failure detector D_i.

We next consider the required properties for the proposed model of the target protocol assuming not only (A1), the guaranteed properties of the uniform

consensus module, but also (A2), the strong completeness of the unreliable failure detectors. As explained in Section 5.1.1, the strong completeness of the failure detectors is expressed in LTL as follows:

$$P_{SC} = \Box(\exists p_i \in \mathcal{P}.(crash[i]) \rightarrow \Diamond\Box(\forall p_j \in \mathcal{P}.(crash[j] \vee S[j].el[i]))).$$

Let P_T, P_J, and P_O respectively denote the properties required for the proposed model to satisfy the termination property, the justification property, and the obligation property provided that (A1) and (A2) hold. P_T, P_J, and P_O are constructed using P_{SC} as follows:

$$\begin{aligned}
P_T &= P_{SC} \rightarrow P_t \\
&= \Box(\exists p_i \in \mathcal{P}.(crash[i]) \rightarrow \Diamond\Box(\forall p_j \in \mathcal{P}.(crash[j] \vee S[j].el[i]))) \\
&\quad \rightarrow \Diamond(\forall p_i \in \mathcal{P}.(fin[i])), \\
P_J &= P_{SC} \rightarrow P_j \\
&= \Box(\exists p_i \in \mathcal{P}.(crash[i]) \rightarrow \Diamond\Box(\forall j \in \mathcal{P}.(crash[j] \vee S[j].el[i]))) \\
&\quad \rightarrow \Box(\exists p_i \in \mathcal{P}.(propose[i] = commit) \rightarrow \forall p_j \in \mathcal{P}.(vote[j] = yes)), \\
P_O &= P_{SC} \rightarrow P_o \\
&= \Box(\exists p_i \in \mathcal{P}.(crash[i]) \rightarrow \Diamond\Box(\forall p_j \in \mathcal{P}.(crash[j] \vee S[j].el[i]))) \\
&\quad \rightarrow \Box(\exists p_i \in \mathcal{P}.(propose[i] = abort) \rightarrow \exists p_j \in \mathcal{P}.(vote[j] \neq yes \vee ex_sus[j])).
\end{aligned}$$

5.3 Modeling in SMV and Specifying CTL Properties

Now we explain the modeling of the target NBAC protocol in SMV. The SMV model is constructed similarly as the Promela model. Figure 5 shows the SMV model for the NBAC protocol in the case of N=2.

The model consists of the module named `node` (Lines 47–92) representing a computing node participating in the protocol and the module named `detector` (Lines 30–45) representing a failure detector. In addition, the module named `channel` (Lines 10–28) is declared to model communication channels for each node.

The main module (Lines 1–8) declares N nodes, `p0`–`p1`, N failure detectors, `d0`–`d1`, and N channels, `c0`–`c1`, as asynchronous processes by using the `process` keyword. In the SMV execution semantics, a process is nondeterministically chosen and executed but is not forced to eventually be chosen. In order to force the processes of nodes and channels to execute infinitely often, we added the declaration '`FAIRNESS running`' as in Lines 28 and 92.

The required CTL properties for the termination, justification, and obligation properties on the SMV model are then constructed in a similar way as the LTL properties given in Section 5.2.

First, the termination property P_T is expressed in CTL as follows:

$$P_T = AG(\exists p_i \in \mathcal{P}.(p_i.state = crash)$$

```
MODULE main
 VAR
  c0 : process channel(0,p0,p1);
  c1 : process channel(1,p0,p1);
  d0 : process detector(0);
  d1 : process detector(1);
  p0 : process node(0,d0,c0);
  p1 : process node(1,d1,c1);

MODULE channel(id,p0,p1)
 VAR
  b_0: 0..2;
  b_1: 0..2;
 ASSIGN
  init(b_0) :=  2;
  init(b_1) :=  2;
  next(b_0) :=
        case
          p0.sent : p0.vote;
          1 : b_0;
        esac;
  next(b_1) :=
        case
          p1.sent : p1.vote;
          1 : b_1;
        esac;

 FAIRNESS running

MODULE detector(id)
 VAR
  _0 : boolean;
  _1 : boolean;
  _ex_sus : boolean;
 ASSIGN
  init(_0) := 0;
  init(_1) := 0;
  next(_0) := {0,1};
  next(_1) := {0,1};
  init(_ex_sus) := 0;
  next(_ex_sus) :=
        case
          _ex_sus : _ex_sus;
          1 : _0 | _1 ;
        esac;

MODULE node(id, d, chan)
 VAR
  state: {sending, waiting, consensus, failed};
  crash: boolean;
  vote: boolean;
  propose: 0..2;
  sent: boolean;

 ASSIGN
  init(state) := sending;
  next(state) :=
        case
          state=sending & crash          : failed;
          state=sending & sent           : waiting;
          state=waiting & crash          : failed;
          state=waiting & propose!=2     : consensus;
          1                              : state;
        esac;

  init(crash) := {0,1};
  next(crash) :=
        case
          !crash & (state=sending | state=waiting) : {0,1};
          1     : crash;
        esac;

  init(vote) := {0,1};
  next(vote) := vote;

  init(sent) := 0;
  next(sent) := case
                  state=sending & !crash : 1;
                  1 : sent;
                esac;

  init(propose) := 2;
  next(propose) :=
        case
          state=waiting & d._ex_sus  : 0;
          state=waiting & chan.b_0=0 : 0;
          state=waiting & chan.b_1=0 : 0;
          state=waiting & chan.b_0=1 & chan.b_1=1   : 1;
          1 : propose;
        esac;

  FAIRNESS running
```

Figure 5: SMV Model for the NBAC Protocol (with N=2).

$$\rightarrow AFAG(\forall p_j \in \mathcal{P}.(p_j.state = crash \vee d_j.\_i)))$$
$$\rightarrow AF(\forall p_i \in \mathcal{P}.(p_i.state = failed \vee p_i.state = consensus))$$

where $d_i.\_j = 1$ means that D_i suspects p_j in the SMV model.

Next, the justification property P_J is expressed in CTL as follows:

$$\begin{aligned} P_J \quad = \quad & AG(\exists p_i \in \mathcal{P}.(p_i.state = crash) \\ & \rightarrow AFAG(\forall p_j \in \mathcal{P}.(p_j.state = crash \vee d_j.\_i))) \\ & \rightarrow AG(\exists p_i \in \mathcal{P}.(p_i.propose = commit) \rightarrow \forall p_j \in \mathcal{P}.(p_j.vote = yes)). \end{aligned}$$

Thirdly, the obligation property is expressed in CTL as follows:

$$\begin{aligned} P_O \quad = \quad & AG(\exists p_i \in \mathcal{P}.(p_i.state = crash) \\ & \rightarrow AFAG(\forall p_j \in \mathcal{P}.(p_j.state = crash \vee d_j.\_i))) \\ & \rightarrow AG(\exists p_i \in \mathcal{P}.(p_i.propose = abort) \rightarrow \exists p_j \in \mathcal{P}.(p_j.vote \neq yes \vee d_j.\_ex\_sus)). \end{aligned}$$

6 Verification of NBAC Protocol

6.1 Model Checking by SPIN and SMV

Whether the target NBAC protocol satisfies the termination, justification, and obligation properties or not can be verified by model checking if the properties P_T, P_J, and P_O hold for our Promela or SMV models given in Section 5.

Figure 6 shows the result of model checking with SPIN whether the termination property P_T holds for the sample Promela model given in Figure 3. The result shows that no error is detected (in Line 13) and thus the target NBAC protocol satisfies the termination property when N=2. When running SPIN, we add the fairness restriction such that a process will be eventually executed if the process has an executable statement whose executability never changes, by using the '`-f`' option (in Line 3). Recall that, for the SMV model, we added the fairness declaration for each process for the same purpose.

For SMV model checking, we used the model checking tool NuSMV [10]. Figure 6 shows the output of NuSMV obtained as a result of checking whether the termination property P_T holds for the sample SMV model given in Figure 5. The verification result is true (in Line 6) and is the same as the case of using SPIN. The NuSMV option '`-r`' (in Line 1) was used to print the numbers of global states and reachable states for the input SMV model.

As a result of checking the three properties P_T, P_J, and P_O, we confirmed that our Promela and SMV models for the NBAC protocol satisfy the termination, justification, and obligation properties.

6.2 Reducing State Space

The main difficulty in verification by model checking is the state explosion problem. To reduce the state space needed for verification, we applied several techniques which will be explained in the following, in addition to the modular

```
% spin -a -N ex.ltl ex.pml
% gcc -o ex.exe pan.c
% time ./ex.exe -a -f
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness enabled)
        invalid end states      - (disabled by never claim)

State-vector 64 byte, depth reached 2937, errors: 0
3.02036e+06 states, stored (1.22808e+07 visited)
5.3665e+07 states, matched
6.59458e+07 transitions (= visited+matched)
      15 atomic steps
hash conflicts: 1.13095e+08 (resolved)

Stats on memory usage (in Megabytes):
217.466 equivalent memory usage for states (stored*(State-vector + overhead))
169.509 actual memory usage for states (compression: 77.95%)
        State-vector as stored = 48 byte + 8 byte overhead
2.097   memory used for hash table (-w19)
0.320   memory used for DFS stack (-m10000)
0.041   other (proc and chan stacks)
0.139   memory lost to fragmentation
171.787 total actual memory usage

unreached in proctype :init:
        (0 of 6 states)
unreached in proctype detector
        line 43, state 24, "-end-"
        (1 of 24 states)
unreached in proctype node
        (0 of 36 states)

real    1m36.217s
user    1m35.910s
sys     0m0.270s
```

Figure 6: Example of the Output of SPIN.

```
% time NuSMV -r ex.smv
*** This is NuSMV 2.3.0 (compiled on Tue Nov 22 01:40:53 UTC 2005)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.
-- specification  AG (((P0 -> AF AG Q0) & (P1 -> AF AG Q1)) -> AF (R0 & R1))  is true
system diameter: 15
reachable states: 112336 (2^16.7775) out of 1.19439e+07 (2^23.5098)

real    0m0.208s
user    0m0.200s
sys     0m0.010s
```

Figure 7: Example of the Output of NuSMV.

modeling and specifying the properties based on the assume-guarantee method described in Section 5.

Appendices A and B respectively show the sample Promela model and the sample SMV model for the target NBAC protocol with N=2 to which we applied the additional modeling techniques for reducing state space.

Let F denote the maximum number of nodes that can be in the failed sate at the same time. The main technique applied is to divide F processes, named `f_node`, that may fail and N-F processes, named `c_node`, that are always operational. We also used the `d_step`-statement and the `atomic`-statement for the Promela model. Using `d_step` and `atomic` statements prevents the statements that are to be executed from being interfered with by other processes.

While the number of global states stored for verification using the Promela model in Figure 3 is 302036 as shown in Line 14 of Figure 6, the one using the improved Promela model in Appendix A is 32225. Similarly, while the number of states for the SMV model in Figure 5 is 11943900 as shown in Line 8 of Figure 7, the one for the model in Appendix B is 3359230.

6.3 Detecting Errors and Diagnostic Counterexample

Here we show how the proposed method can detect errors for a given target protocol, using an example.

Recall that the NBAC protocol in Figure 2 requires that the failure detectors satisfy the strong completeness property. For example, consider that there is another NBAC protocol, $\mathcal{P}'$, which does not assume any completeness property for the failure detectors. Whether the protocol $\mathcal{P}'$ satisfies the termination, justification, and obligation properties or not can be verified by checking if the properties P_t, P_j, and P_o given in Section 5 hold for our model.

The result of checking the termination property for the protocol $\mathcal{P}'$ is false while the results of checking the justification and obligation properties for $\mathcal{P}'$ are true. Figure 8 shows a counterexample trace output from SPIN when checking P_t for the Promela model in Appendix A. By analyzing the trace, one can specify the following error of the protocol $\mathcal{P}'$ to violate the termination: After a node p_i crashes before sending its vote, the other node p_j keeps waiting until receiving the vote of p_i and hence p_j cannot make its decision.

7 Experiment

In the experiment, we checked the termination, justification, obligation properties of the NBAC protocol with $2 \leq N \leq 6$. For comparison, we constructed both of the Promela models and the SMV models for the target NBAC protocol and verified the required properties using model checking tools SPIN and NuSMV respectively. The obtained verification results are true for all example cases, and thus we can confirm that our models for the NBAC protocol satisfy the termination, justification, obligation properties for the cases tested.

```
Starting :init: with pid 0
spin: couldn't find claim (ignored)
Starting f_node with pid 2
  2:    proc  0 (:init:) line  18 "ex2.pml" (state 1)   [(run f_node(0))]
Starting c_node with pid 3
  3:    proc  0 (:init:) line  19 "ex2.pml" (state 2)   [(run c_node(1))]
Starting detector with pid 4
  4:    proc  0 (:init:) line  20 "ex2.pml" (state 3)   [(run detector(0))]
Starting detector with pid 5
  5:    proc  0 (:init:) line  21 "ex2.pml" (state 4)   [(run detector(1))]
  7:    proc  4 (detector) line  36 "ex2.pml" (state 1) [(1)]
  8:    proc  4 (detector) line  28 "ex2.pml" (state 2) [(1)]
  8:    proc  4 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]

  ...

  <<<<<START OF CYCLE>>>>>
 83:    proc  4 (detector) line  36 "ex2.pml" (state 1) [(1)]
 84:    proc  4 (detector) line  28 "ex2.pml" (state 2) [(1)]
 84:    proc  4 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
 86:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
 87:    proc  3 (detector) line  28 "ex2.pml" (state 2) [(1)]
 87:    proc  3 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
 89:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
 90:    proc  3 (detector) line  29 "ex2.pml" (state 4) [(1)]
 90:    proc  3 (detector) line  29 "ex2.pml" (state 5) [S[i].el[0] = 1]
 90:    proc  3 (detector) line  29 "ex2.pml" (state 6) [ex_sus[i] = 1]
 92:    proc  3 (detector) line  37 "ex2.pml" (state 12)        [(1)]
 93:    proc  3 (detector) line  29 "ex2.pml" (state 15)        [(1)]
 93:    proc  3 (detector) line  29 "ex2.pml" (state 16)        [S[i].el[1] = 1]
 93:    proc  3 (detector) line  29 "ex2.pml" (state 17)        [ex_sus[i] = 1]
 95:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
 96:    proc  3 (detector) line  28 "ex2.pml" (state 2) [(1)]
 96:    proc  3 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
 98:    proc  4 (detector) line  36 "ex2.pml" (state 1) [(1)]
 99:    proc  4 (detector) line  28 "ex2.pml" (state 2) [(1)]
 99:    proc  4 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
101:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
102:    proc  3 (detector) line  29 "ex2.pml" (state 4) [(1)]
102:    proc  3 (detector) line  29 "ex2.pml" (state 5) [S[i].el[0] = 1]
102:    proc  3 (detector) line  29 "ex2.pml" (state 6) [ex_sus[i] = 1]
104:    proc  3 (detector) line  37 "ex2.pml" (state 12)        [(1)]
105:    proc  3 (detector) line  28 "ex2.pml" (state 13)        [(1)]
105:    proc  3 (detector) line  28 "ex2.pml" (state 14)        [S[i].el[1] = 0]
107:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
108:    proc  3 (detector) line  28 "ex2.pml" (state 2) [(1)]
108:    proc  3 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
110:    proc  4 (detector) line  36 "ex2.pml" (state 1) [(1)]
111:    proc  4 (detector) line  28 "ex2.pml" (state 2) [(1)]
111:    proc  4 (detector) line  28 "ex2.pml" (state 3) [S[i].el[0] = 0]
113:    proc  3 (detector) line  36 "ex2.pml" (state 1) [(1)]
114:    proc  3 (detector) line  29 "ex2.pml" (state 4) [(1)]
114:    proc  3 (detector) line  29 "ex2.pml" (state 5) [S[i].el[0] = 1]
114:    proc  3 (detector) line  29 "ex2.pml" (state 6) [ex_sus[i] = 1]
116:    proc  3 (detector) line  37 "ex2.pml" (state 12)        [(1)]
117:    proc  3 (detector) line  29 "ex2.pml" (state 15)        [(1)]
117:    proc  3 (detector) line  29 "ex2.pml" (state 16)        [S[i].el[1] = 1]
117:    proc  3 (detector) line  29 "ex2.pml" (state 17)        [ex_sus[i] = 1]
119:    proc  3 (detector) line  37 "ex2.pml" (state 12)        [(1)]
120:    proc  3 (detector) line  28 "ex2.pml" (state 13)        [(1)]
120:    proc  3 (detector) line  28 "ex2.pml" (state 14)        [S[i].el[1] = 0]
spin: trail ends after 120 steps
#processes: 5
                crash[0] = 1
                crash[1] = 0
                S[0].el[0] = 1
                S[0].el[1] = 0
                S[1].el[0] = 0
                S[1].el[1] = 0
                ex_sus[0] = 1
                ex_sus[1] = 0
                vote[0] = 1
                vote[1] = 1
                propose[0] = 0
                propose[1] = 0
                fin[0] = 1
                fin[1] = 0
                queue 1 (ch2p[0]): [1]
                queue 2 (ch2p[1]):
120:    proc  4 (detector) line  35 "ex2.pml" (state 23)
120:    proc  3 (detector) line  35 "ex2.pml" (state 23)
120:    proc  2 (c_node) line  77 "ex2.pml" (state 31)
120:    proc  1 (f_node) line  71 "ex2.pml" (state 42) <valid end state>
120:    proc  0 (:init:) line  23 "ex2.pml" (state 6) <valid end state>
5 processes created
```

Figure 8: Example of a Counterexample Trace of SPIN.

Table 2: Experimental Performance Results.

			SPIN		SMV	
	N	F	# states	Time	# sates	Time
Termination	2	1	32225	0m2.110s	3.35923E+06	0m0.130s
	3	1	7.40305E+07	154m59.080s	9.4037E+11	10m37.940s
		2	NA	NA	1.88074E+12	11m0.940s
	4	1	NA	NA	NA	NA
		2	NA	NA	NA	NA
		3	NA	NA	NA	NA
Justification	2	1	8209	0m0.100s	3.35923E+06	0m0.030s
	3	1	2.54525E+07	7m59.350s	9.4037E+11	0m1.040s
		2	3.05771E+07	11m30.380s	1.88074E+12	0m1.770s
	4	1	NA	NA	9.47676E+18	14m23.910s
		2	NA	NA	1.89535E+19	26m16.560s
		3	NA	NA	3.79071E+19	34m55.360s
	5	1	NA	NA	NA	NA
Obligation	2	1	8209	0m0.140s	3.35923E+06	0m0.020s
	3	1	2.54525E+07	9m10.960s	9.4037E+11	0m0.050s
		2	3.05771E+07	12m57.870s	1.88074E+12	0m0.030s
	4	1	NA	NA	9.47676E+18	0m1.030s
		2	NA	NA	1.89535E+19	0m1.050s
		3	NA	NA	3.79071E+19	0m1.040s
	5	1	NA	NA	-	60m33.920s
		2	NA	NA	-	61m55.320s
		3	NA	NA	-	61m21.730s
		4	NA	NA	-	61m24.250s
	6	1	NA	NA	NA	NA

Table 2 shows the performance results in terms of the number of states stored for the Promela and SMV models and the execution time needed for model checking by SPIN and NuSMV. Data was collected using a Linux workstation with an Intel Xeon 3.0 GHz and 4 GB memory. In the table, '*NA*' denotes that data was not collected since the verification was not completed for the memory shortage, while '-' denotes that data was not collected since the enumeration of states was not completed for the memory shortage although the verification was completed successfully.

In Table 2, for the all cases, the number of states required for the Promela models is less than that for the SMV models. However, the time needed for the verification by SPIN was longer than the one by NuSMV. We conjecture that the reason for this is that SMV can represent the state space more compactly by using a symbolic representation and thus can reduce the memory and time required for verification compared to SPIN.

8 Conclusion

This paper proposed a method for model checking the modular-structured non-blocking atomic commitment protocol based on consensus and unreliable failure detectors. In the proposed method, we adopted an assume-guarantee approach as follows: We constructed a finite state model for the target protocol by composing the behavior models for computing nodes and failure detectors. At the same time, we constructed temporal logic formulae expressing the termination, justification, and obligation properties for the model, assuming the guaranteed properties by the consensus and the failure detectors. Our approach can improve a verification speed and reduce the state space needed for model checking. In this paper, we presented the experimental verification results using the two model checking tools SPIN and SMV. We conjecture that our approach can be generalized to other modular-structured fault-tolerant distributed protocols based on consensus and failure detectors.

Acknowledgment

This work was supported by Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan (No. 18049055).

References

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

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):245–267, 1996.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[5] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. *Proceedings of the 9th WDAG, Lecture Notes in Computer Science*, (972):87–100, 1995.

[6] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, 2001.

[7] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[8] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science - Modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[10] NuSMV. http://nusmv.irst.itc.it/.

[11] M. Raynal and M. Singhal. Mastering agreement problems in distributed systems. *IEEE Software*, 18(4):40–47, 2001.

[12] D. Skeen. Nonblocking commit protocols. In *Proceedings of ACM SIGMOD International Conf. Management of Data*, pages 133–142, 1981.

Appendix

A. Modified Promela model

```
# define N 2 /* # of nodes */
# define F 1 /* maximun # of faulty nodes */
# define C 1 /* N-F */

bool crash[N];
typedef Array{
  bool el[N]
};
Array S[N];
bool ex_sus[N];
bool vote[N];
bool propose[N];
bool fin[N];
chan ch2p[N] = [N] of {bool};

init{
  atomic{
    run f_node(0);
    run c_node(1);
    run detector(0);
    run detector(1);
  }
}

inline guess(j){
  atomic{
    if
      :: true -> S[i].el[j] = 0
      :: true -> S[i].el[j] = 1; ex_sus[i] = 1
    fi;
  }
}

proctype detector(byte i){
  do
    :: atomic{ true -> guess(0) }
    :: atomic{ true -> guess(1) }
  od
}

inline voting(){
  atomic{
    if
      :: true -> vote[i] = 1
      :: true -> vote[i] = 0;
    fi
  }
}

inline sending(){
  atomic{
    ch2p[0]!vote[i];
    ch2p[1]!vote[i]
  }
}

proctype f_node(byte i) {
  byte yesnum;
  bool flag;

  voting(); flag=1;
  do
    :: d_step{ flag -> sending(); flag=0 }
    :: d_step{ !flag && ch2p[i]?[1] && !ex_sus[i] -> ch2p[i]?_; yesnum++ }
    :: atomic{ !flag && ch2p[i]?[0] -> ch2p[i]?_; propose[i] = 0; break }
    :: atomic{ !flag && ex_sus[i] -> propose[i] = 0; break }
    :: atomic{ !flag && yesnum==N -> propose[i] = 1; break }
    :: atomic{ true -> crash[i] = 1; break }
  od;
  fin[i]=1
}

proctype c_node(byte i) {
  byte yesnum;

  d_step{ voting(); sending(); }
  do
    :: d_step{ ch2p[i]?[1] && !ex_sus[i] -> ch2p[i]?_; yesnum++ }
    :: atomic{ ch2p[i]?[0] -> ch2p[i]?_; propose[i] = 0; break }
    :: atomic{ ex_sus[i] -> propose[i] = 0; break }
    :: atomic{ yesnum==N -> propose[i] = 1; break }
  od;
  fin[i]=1
}
```

B. Modified SMV model

```
MODULE main
 VAR
  d0 : process detector(0);
  d1 : process detector(1);
  c0 : process channel(0,p0,p1);
  c1 : process channel(1,p0,p1);
  p0 : process f_node(0,d0,c0);
  p1 : process c_node(1,d1,c1);

 DEFINE
  P0 := p0.crash;
  Q0 := d1._0;
  R0 := (p0.state=fin);
  R1 := (p1.state=fin);
 SPEC AG( (P0 -> AF AG Q0) -> (AF (R0 & R1)) )

MODULE channel(id,p0,p1)
 VAR
  b_0: 0..2;
  b_1: 0..2;
 ASSIGN
  init(b_0) :=  2;
  init(b_1) :=  2;
  next(b_0) :=
        case
          p0.sent : p0.vote;
          1 : b_0;
        esac;
  next(b_1) :=
        case
          p1.sent : p1.vote;
          1 : b_1;
        esac;

 FAIRNESS running

MODULE detector(id)
 VAR
  _0 : boolean;
  _1 : boolean;
  _ex_sus : boolean;
 ASSIGN
  init(_0) := 0;
  init(_1) := 0;
  next(_0) := {0,1};
  next(_1) := {0,1};
  init(_ex_sus) := 0;
  next(_ex_sus) :=
        case
          _ex_sus : _ex_sus;
          1 : _0 | _1 ;
        esac;

MODULE f_node(id, d, chan)
 VAR
  state: {sending, waiting, fin};
  crash: boolean;
  vote: boolean;
  propose: 0..2;
  sent: boolean;

 ASSIGN
  init(state) := sending;
  next(state) :=
        case
          state=sending & crash          : fin;
          state=sending & sent           : waiting;
          state=waiting & crash          : fin;
          state=waiting & propose!=2     : fin;
          1                              : state;
        esac;

  init(crash) :=
        case
          id=0  : {0,1};
          1     : 0;
        esac;
  next(crash) :=
        case
          !crash & (state=sending | state=waiting) : {0,1};
          1     : crash;
        esac;

  init(vote) := {0,1};
  next(vote) := vote;

  init(sent) := 0;
  next(sent) := case
                  state=sending & !crash : 1;
                  1 : sent;
                esac;

  init(propose) := 2;
  next(propose) :=
        case
          state=waiting & d._0   : 0;
          state=waiting & d._1   : 0;
          state=waiting & chan.b_0=0 : 0;
          state=waiting & chan.b_1=0 : 0;
          state=waiting & chan.b_0=1 & chan.b_1=1   : 1;
          1 : propose;
        esac;

  FAIRNESS running

MODULE c_node(id, d, chan)
 VAR
  state: {sending, waiting, fin};
  vote: boolean;
  propose: 0..2;
  sent: boolean;

 ASSIGN
  init(state) := sending;
  next(state) :=
        case
          state=sending & sent           : waiting;
          state=waiting & propose!=2     : fin;
          1                              : state;
        esac;

  init(vote) := {0,1};
  next(vote) := vote;

  init(sent) := 0;
  next(sent) := case
                  state=sending : 1;
                  1 : sent;
                esac;

  init(propose) := 2;
  next(propose) :=
        case
          state=waiting & d._0   : 0;
          state=waiting & d._1   : 0;
          state=waiting & chan.b_0=0 : 0;
          state=waiting & chan.b_1=0 : 0;
          state=waiting & chan.b_0=1 & chan.b_1=1 : 1;
          1 : propose;
        esac;

  FAIRNESS running
```

モジュラー構造の非ブロッキング原子コミットプロトコルのモデル検査
(in English)
(算譜科学研究速報)
発行日：2006 年 11 月 2 日
編集・発行：独立行政法人産業技術総合研究所システム検証研究センター
同連絡先：〒563-8577 大阪府池田市緑丘 1-8-31
e-mail：informatics-inquiry@m.aist.go.jp

Model Checking a Modular-Structured Nonblocking Atomic Commitment Protocol for Asynchronous Distributed Systems
(Programming Science Technical Report)
Nov. 2, 2006
Research Center for Verification and Semantics (CVS)
National Institute of Advanced Industrial Science and Technology (AIST)
1-8-31 Midorigaoka, Ikeda, Osaka, 563-8577, Japan
e-mail: informatics- inquiry@m.aist.go.jp