Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder

Vienna University of Technology (TU Wien)

Abstract. Fault-tolerant distributed algorithms are central for building reliable, spatially distributed systems. In order to ensure that these algorithms actually make systems more reliable, we must ensure that these algorithms are actually correct. Unfortunately, model checking state-of-the-art fault-tolerant distributed algorithms (such as Paxos) is currently out of reach except for very small systems.

In order to be eventually able to automatically verify such fault-tolerant distributed algorithms also in larger systems, several problems have to be addressed. In this paper, we consider modeling and verification of fault-tolerant algorithms that basically only contain threshold guards to control the flow of the algorithm. As threshold guards are widely used in fault-tolerant distributed algorithms (and also in Paxos) efficient methods to handle them bring us closer to the above mentioned goal.

As case study we use the reliable broadcasting algorithm by Srikanth and Toueg that tolerates even Byzantine faults. We show how one can model this basic fault-tolerant distributed algorithm in Promela such that safety and liveness properties can be efficiently verified in Spin. We provide experimental data also for other distributed algorithms.

1 Introduction

Even formally verified computer systems are subject to power outages, electrical wear-out, bit-flips in memory due to ionizing particle hits, etc. that may easily cause system failures. Replication is a classic approach to ensure that a computer system is fault-tolerant, i.e., continues to correctly perform its task even if some components fail. The basic idea is to have multiple computers instead of a single one (that would constitute a single point of failure), and ensure that the replicated computers coordinate, and for instance in the case of replicated databases, store the same information. Ensuring that all computers agree on the same information is non-trivial due to several sources of non-determinism, namely, faults, uncertain message delays, and asynchronous computation steps.

To address these issues, fault-tolerant distributed algorithms for state-machine replication were introduced many years ago [33]. As they are designed to increase the reliability of computing systems, it is crucial that these algorithms are indeed correct, i.e., satisfy their specifications. Due to the various sources of non-determinism, however, it is very easy to make mistakes in the correctness arguments for fault-tolerant distributed algorithms. As a consequence, they are

very natural candidates for model checking. Still, model checking fault-tolerant distributed algorithms is particularly challenging due to the following reasons:

(i) Due to their inherent concurrency and the many sources of non-determinism, fault-tolerant distributed algorithms suffer from combinatorial explosion both in the state-space and in the number of legal behaviors. Moreover, distributed algorithms usually involve parameters such as the system size n and the maximum number of faulty components t.

(ii) Correctness and even solvability of problems like distributed agreement depend critically upon assumptions on the environment, in particular, degree of concurrency, message delays, and failure models; e.g., guaranteeing correct execution is impossible if there is no restriction on the number of faulty components in the system and/or the way how they may fail.

(iii) There is no commonly agreed-upon distributed computing model, but rather many variants, which differ in (sometimes subtle) details such as atomicity of a computing step. Moreover, distributed algorithms are usually described in pseudocode, typically using different (alas unspecified) pseudocode languages, which obfuscates the relation to the underlying computing model.

A central and important goal of our recent work is hence to initiate a systematic study of distributed algorithms from a verification point of view, in a way that does not betray the fundamentals of distributed algorithms. Experience tells that this has not always been observed in the past: The famous bakery algorithm [22] is probably the most striking example from the literature where wrong specifications have been verified or wrong semantics have been considered: Many papers in formal methods have verified the correctness of the bakery algorithm as an evidence for their practical applicability. Viewed from a distributed algorithms perspective, however, most of these papers missed the fact that the algorithm does not require atomic registers but rather safe registers only [23] — a subtle detail that is admittedly difficult to extract from the distributed algorithms literature for non-experts. Still, compared to state-of-the-art fault-tolerant distributed algorithms — and even the algorithms considered in this paper — the bakery algorithm rests on a quite simple computational model, which shows the need for a structured approach to handle distributed algorithms.

Contributions. In this paper, we present a structured approach for modeling an important family of fault-tolerant distributed algorithms, namely, threshold-guarded distributed algorithms discussed in Section 2. As threshold-guarded commands are omnipresent in this domain, our work is an important step towards the goal of verifying state-of-the-art fault-tolerant distributed algorithms. In Section 3, we obtain models of distributed algorithms expressed in slightly extended Promela [20] to capture the notions required to fully express fault-tolerant distributed algorithms and their environments, including resilience conditions involving parameters like n and t, fairness conditions, and atomicity assumptions. This formalization allows us to (i) instantiate system instances for different system sizes in order to perform explicit state model checking using Spin as discussed in Section 4, and (ii) build a basis for our parameterized model checking technique based on parametric interval abstraction discussed in [21].

Using our approach, we can already formalize and model check several basic fault-tolerant distributed algorithms for fixed parameters, i.e., numbers of processes and faults. These algorithms include several variants of the classic asynchronous broadcasting algorithm from [34] under various fault assumptions, the broadcasting algorithm from [6] tolerating Byzantine faults, the classic broadcasting algorithm found, e.g., in [9], that tolerates crash faults, as well as a condition-based consensus algorithm [27] that also tolerates crash faults.

This captures the most interesting problems that are solvable [16] by distributed algorithms running in a purely asynchronous environment with faults. Our verification results build a corner stone for the verification of more advanced fault-tolerant distributed algorithms that require more restricted environments; for instance, the algorithms in [13,9,26,37,10,18] use threshold-guarded commands as a building block.

2 Threshold-guarded distributed algorithms

Processes, which constitute the distributed algorithms we consider, exchange messages, and change their state predominantly based on the received messages. In addition to the standard execution of actions, which are guarded by some predicate on the local state, most basic distributed algorithms (cf. [24,3]) add existentially or universally guarded commands involving received messages:

```
if received <m>
   from some process
then action(m);
```

(a) existential guard

```
if received <m>
   from all processes
then action(m);
```

(b) universal guard

Depending on the content of the message `<m>`, the function `action` performs a local state transition and possibly sends messages to one or more processes. Such constructs can be found, e.g., in (non-fault-tolerant) distributed algorithms for constructing spanning trees, flooding, mutual exclusion, or network synchronization [24]. Understanding and analyzing such distributed algorithms is far from being trivial, which is due to the partial information on the global state present in the local state of a process. After all, real processors execute at different and varying speeds, and the end-to-end message delays also vary considerably. Viewed from the global perspective, this results in considerable non-determinism of the executions of a distributed system.

An additional source of non-determinism are faults. In order to shed some light on the difficulties facing a distributed algorithm in the presence of faults, consider Byzantine faults [28], which allow a faulty process to behave arbitrarily: Faulty processes may fail to send messages, send messages with erroneous values, or even send conflicting information to different processes. In addition, faulty processes may even collaborate in order to increase their adverse power.

Fault-tolerant distributed algorithms work in the presence of such faults and provide some “higher level” service. In case of distributed agreement (consensus),

e.g., it should be guaranteed that all non-faulty processes compute the same result even if some processes fail. Fault-tolerant distributed algorithms are hence used for increasing the system-level reliability of distributed systems [30].

If one tries to build such a fault-tolerant distributed algorithm using the construct of Example (a) in the presence of Byzantine faults, the (local state of the) receiver process would be corrupted if the received message `<m>` originates in a faulty process. A faulty process could hence contaminate a correct process. On the other hand, if one tried to use the construct of Example (b), a correct process would wait forever (starve) when a faulty process omits to send the required message. To overcome those problems, fault-tolerant distributed algorithms typically require assumptions on the maximum number of faults, and employ suitable thresholds for the number of messages which can be expected to be received by correct processes. Assuming that the system consists of n processes among which at most t may be faulty, *threshold-guarded commands* such as the following are typically used by fault-tolerant distributed algorithms:

```
if received <m> from n-t distinct processes
then action(m);
```

Assuming that thresholds are functions of the parameters n and t, threshold guards are a just generalization of quantified guards as given in Examples (a) and (b): In the above command, a process waits to receive $n - t$ messages from distinct processes. As there are at least $n - t$ correct processes, the guard cannot be blocked by faulty processes, which avoids the problems of Example (b). In the distributed algorithms literature, one finds a variety of different thresholds: Typical numbers are $\lceil n/2+1 \rceil$ (for majority [13,27]), $t+1$ (to wait for a message from at least one correct process [34,13]), or $n - t$ (in the Byzantine case [34,2] to wait for at least $t + 1$ messages from correct processes, provided $n > 3t$).

In the setting of Byzantine fault tolerance, it is important to note that the use of threshold-guarded commands implicitly rests on the assumption that a receiver can distinguish messages from different senders. In practice, this can be achieved e.g. by using point-to-point links between processes or by message authentication. What is important here is that Byzantine faulty processes are only allowed to exercise control on their own messages and computations, but not on the messages sent by other processes and the computation of other processes.

Reliable broadcast and related specifications. The specifications considered in the area of fault tolerance differ from more classic areas, such as concurrent systems where dining philosophers and mutual exclusion are central problems. For the latter, one is typically interested in local properties, e.g., if a philosopher i is hungry, then i eventually eats. Intuitively, dining philosophers requires us to trace indexed processes along a computation, e.g., $\forall i.\ \mathbf{G}\,(\text{hungry}_i \rightarrow (\mathbf{F}\,\text{eating}_i))$, and thus to employ *indexed* temporal logics for specifications [7,11,12,14].

In contrast, fault-tolerant distributed algorithms are typically used to achieve certain *global* properties. Reliable broadcast is an ongoing "system service" with the following informal specification: Each process i may invoke a primitive called broadcast by calling $bcast(i, m)$, where m is a unique message content. Processes

may deliver a messages by invoking *accept*(i, m) for different process and message pairs (i, m). The goal is that all correct processes invoke *accept*(i, m) for the same set of (i, m) pairs, with the requirement that all messages broadcast by a correct process are accepted by all correct processes, and that *accept*(i, m) is not invoked if i is correct and i did not invoke *bcast*(i, m). Our case study is to verify that the algorithm from [34] implements these primitives on top of point-to-point channels, in the presence of Byzantine faults. In [34], the instances for different (i, m) pairs do not interfere. Therefore, we will not consider i and m (since in [34] the domain of m is not specified, some restrictions on m would be required anyway to make the algorithm amenable to finite state model checking). Rather, we distinguish the different kinds of invocations of *bcast*(i, m) that may occur, e.g., the cases where the invoking process is faulty or correct. Depending on the initial state, we then have to check whether every/no correct process accepts. To capture this kind of properties, we have to trace only existentially or universally quantified properties, e.g., part of the broadcast specification (relay) [34] states that if some correct process accepts a message, then all (correct) processes accept the message, that is, $(\mathbf{G}\,(\exists i.\ \mathrm{accept}_i)) \rightarrow (\mathbf{F}\,(\forall j.\ \mathrm{accept}_j))$.

We are therefore considering a temporal logic where the *quantification over processes is restricted to propositional formulas.* We will need two kinds of quantified propositional formulas that consider (i) the finite control state modeled as a single status variable sv, and (ii) the possible unbounded data. We introduce the set AP_{SV} that contains propositions that capture comparison against some status value Z from the set of all control states, i.e., $[\forall i.\ sv_i = Z]$ and $[\exists i.\ sv_i = Z]$.

This allows us to express specifications of distributed algorithms. To express the mentioned relay property, we identify the status values where a process has accepted the message. We may quantify over all processes as we only explicitly model those processes, which are restricted in their internal behavior, that is, correct or benign faulty processes. More severe faults that are unrestricted in their internal behavior (e.g., Byzantine faults) are modeled via non-determinism in message passing. For a detailed discussion see Section 3.

In order to express comparison of data variables, we add a set of atomic propositions AP_D that capture comparison of data variables (integers) x, y, and constant c; AP_D consists of propositions of the form $[\exists i.\ x_i + c < y_i]$.

The labeling function of a system instance is then defined naturally as disjunction or conjunction over all process indices; cf. [21] for complete definitions.

Given an $\mathsf{LTL} \setminus \mathsf{X}$ formula ψ over AP_D expressing justice [29], an $\mathsf{LTL} \setminus \mathsf{X}$ specification φ over AP_{SV}, a process description P in PROMELA, and the number of (correct) processes N, the problem is to verify whether

$$\underbrace{P \parallel P \parallel \cdots \parallel P}_{N\,times} \models \psi \rightarrow \varphi.$$

3 Threshold-guarded distributed algorithms in Promela

Algorithm 1 is our case study for which we also provide a complete PROMELA implementation later in Figure 4. To explain how we obtain this implementation,

Algorithm 1 Core logic of the broadcasting algorithm from [34].

Code for processes i if it is correct:

Variables

1: $v_i \in \{\text{FALSE}, \text{TRUE}\}$
2: $\text{accept}_i \in \{\text{FALSE}, \text{TRUE}\} \leftarrow \text{FALSE}$

Rules

3: **if** v_i **and** not sent $\langle$echo$\rangle$ before **then**
4: *send* $\langle$echo$\rangle$ to all;
5: **if** *received* $\langle$echo$\rangle$ from at least $t+1$ *distinct* processes
and not sent $\langle$echo$\rangle$ before **then**
6: *send* $\langle$echo$\rangle$ to all;
7: **if** *received* $\langle$echo$\rangle$ from at least $n-t$ *distinct* processes **then**
8: $\text{accept}_i \leftarrow \text{TRUE}$;

we proceed in three steps where we first discuss asynchronous distributed algorithms in general, then explain our encoding of message passing for threshold-guarded fault-tolerant distributed algorithms. Algorithm 1 belongs to this class, as it does not distinguish messages according to their senders, but just counts received messages, and performs state transitions depending on the number of received messages; e.g. line 7. Finally we encode the control flow of Algorithm 1. The rationale of the modeling decisions are that the resulting Promela model (i) captures the assumptions of distributed algorithms adequately, and (ii) allows for efficient verification either using explicit state enumeration (as discussed in this paper) or by abstraction as discussed in [21]. After discussing the modeling of distributed algorithms, we will provide the specifications in Section 3.4.

3.1 Computational model for asynchronous distributed algorithms

We recall the standard assumptions for asynchronous distributed algorithms. As mentioned in the introduction, a system consists of n processes out of which at most t may be faulty. When considering a fixed computation, we denote by f the actual number of faulty processes. Note that f is not "known" to the correct processes. It is assumed that $n > 3t \wedge f \leq t \wedge t > 0$. As these parameters do not change during a run, they can be encoded as constants in Promela. Correct processes follow the algorithm, in that they take steps that correspond to the algorithm description. Between every pair of processes, there is a bidirectional link over which messages are exchanged. A link contains two message buffers, each being the receive buffer of one of the incident processes.

A step of a correct process is *atomic* and consists of the following three parts. (i) The process possibly receives a message. A process is not forced to receive a message even if there is one in its buffer [16]. (ii) Then, it performs a state transition depending on its current state and the received message. (iii) Finally, a process may send at most one message to each process, that is, it puts a message in the buffer of the other processes.

Computations are asynchronous in that the steps can be arbitrarily interleaved, provided that each correct process takes an infinite number of steps.[1] Moreover, if a message m is put into a process p's buffer, and p is correct, then m is eventually received. This property is called *reliable communication.*

From the above discussion we observe that buffers are required to be unbounded, and therefore sending is non-blocking. Further, receiving is non-blocking even if no message has been sent to the process. If we assume that for each message type, each correct process sends at most one message in each run (as in Algorithm 1), non-blocking send can in principle natively be encoded in Promela using message channels. In principle, non-blocking receive also can be implemented in Promela, but it is not a basic construct. We discuss the modeling of message passing in more detail in Section 3.2.

Fault types. In our case study Algorithm 1 we consider *Byzantine* faults, that is, faulty processes are not restricted, except that they have no influence on the buffers of links to which they are not incident. Below we also consider restricted failure classes: *omission faults* follow the algorithm but may fail to send some messages, *crash faults* follow the algorithm but may prematurely stop running. Finally, *symmetric faults* need not follow the algorithm, but if they send messages, they send them to all processes. (The latter restriction does not apply to Byzantine faults which may send conflicting information to different processes).

3.2 Efficient encoding of message passing

In threshold-guarded distributed algorithms, the processes (i) count how many messages of the same type they have received from *distinct* processes, and change their states depending on this number, (ii) always send to *all* processes (including the process itself), and (iii) send messages only for a fixed number of types (only messages of type $\langle$echo$\rangle$ are sent in Algorithm 1).

Fault-free communication. We discuss in the following that one can model such algorithms in a way that is more efficient in comparison to a straight forward implementation with Promela channels. In our final modeling we have an approach that captures both message passing and the influence of faults on correct processes. However, in order to not clutter the presentation we start our discussion by considering communication between correct processes only (i.e., $f = 0$), and add faults later in this section.

In the following code examples we show a straight-forward way to implement "received $\langle$echo$\rangle$ from at least x distinct processes" and "send $\langle$echo$\rangle$ to all" using Promela channels: We declare an array `p2p` of n^2 channels, one per a pair of processes, and then we declare an array `rx` to record that at most one $\langle$echo$\rangle$ message from a process j is received by a process i:

[1] Algorithm 1 has runs that never accept and are infinite. Conceptually, the standard model requires that processes executing terminating algorithms loop forever in terminal states [24].

```
mtype = { ECHO }; /* one message type */
chan p2p[NxN] = [1] of { mtype }; /* channels of capacity 1 */
bit  rx[NxN]; /* a bit map to implement "distinct" */
active[N] proctype STBcastChan() {
  int i, nrcvd = 0; /* nr. of echoes */
```

Then, the receive code iterates over n channels: for non-empty channels it received an ⟨echo⟩ message or not, and empty channels are skipped; if a message is received, the channel is marked in `rx`:

```
  i = 0; do
    :: (i < N) && nempty(p2p[i * N + _pid]) ->
      p2p[i * N + _pid]?ECHO; /* retrieve a message */
      if
        :: !rx[i * N + _pid] ->
          rx[i * N + _pid] = 1; /* mark the channel */
          nrcvd++; break; /* receive at most one message */
        :: rx[i * N + _pid];   /* ignore duplicates */
      fi; i++;
    :: (i < N) ->
      i++;   /* channel empty or postpone reception */
    :: i == N -> break;
  od
```

Finally, the sending code also iterates over n channels and sends on each:

```
  for (i : 1 .. N) { p2p[_pid * N + i]!ECHO; }
```

Recall that threshold-guarded algorithms have specific constraints: messages from all processes are processed uniformly; every message is carrying only a message type without a process identifier; each process sends a message to all processes in no particular order. This suggests a simpler modeling solution. Instead of using message passing directly, we keep only the numbers of sent and received messages in integer variables:

```
int nsnt; /* one shared variable per a message type */
active[N] proctype STBcast() {
  int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes */
  ...
step: atomic {
    if /* receive one more echo */
      :: (next_nrcvd < nsnt) ->
        next_nrcvd = nrcvd + 1;
      :: next_nrcvd = nrcvd; /* or nothing */
    fi;
    ...
    nsnt++; /* send echo to all */
  }
```

As one process step is executed atomically (indivisibly), concurrent reads and updates of *nsnt* are not a concern to us. Note that the presented code is based on the assumption that each correct process sends at most one message.

```
active[F] proctype Byz() {
step: atomic {
  i = 0; do
   :: i < N -> sendTo(i); i++;
   :: i < N -> i++; /* some */
   :: i == N -> break;
  od
 }; goto step;
}

active[F] proctype Omit() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 if :: correctCodeSendsAll ->
   i = 0; do
    :: i < N -> sendTo(i); i++;
    :: i < N -> i++; /* omit */
    :: i == N -> break;
   od
   :: skip;
 fi
 }; goto step;
}
```

```
active[F] proctype Symm() {
step: atomic {
  if
   :: /* send all */
     for (i : 1 .. N)
     { sendTo(i); }
   :: skip; /* or none */
  fi
 }; goto step;
}

active[F] proctype Clean() {
step: atomic {
 /* receive as a correct */
 /* compute as a correct */
 /* send as a correct one */
 };
 if
    :: goto step;
    :: goto crash;
 fi;
crash:
}
```

Fig. 1: Modeling faulty processes explicitly: Byzantine (Byz), symmetric (Symm), omission (Omit), and clean crashes (Clean)

We show how to enforce this assumption when discussing the control flow of our implementation of Algorithm 1 below.

Recall that in asynchronous distributed systems one assumes communication fairness, that is, every message sent is eventually received. The statement $\exists i.\ rcvd_i < nsnt_i$ describes a global state where messages are still in transit. It follows that a formula ψ defined by

$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_i < nsnt_i] \qquad \text{(RelComm)}$$

states that the system periodically delivers all messages sent by (correct) processes. We are thus going to add such fairness requirements to our specifications.

Faulty processes. In Figure 1 we show how one can model the different types of faults discussed above using channels. The implementations are direct consequences of the fault description given in Section 3.1. Figure 2 shows how the impact of faults on processes following the algorithm can be implemented in the shared memory implementation of message passing. Note that in contrast to Figure 1, the processes in Figure 2 are *not* the faulty ones, but correct ones whose variable `next_nrcvd` is subject to non-deterministic updates that corresponds

```
/* N > 3T ∧ T ≥ F ≥ 0 */
active[N-F] proctype ByzI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + F) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N > 2T ∧ T ≥ Fp ≥ Fs ≥ 0 */
active[N-Fp] proctype SymmI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt + Fs) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N > 2T ∧ T ≥ F ≥ 0 */
active[N] proctype OmitI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

```
/* N ≥ T ∧ T ≥ Fc ≥ Fnc ≥ 0 */
active[N] proctype CleanI() {
step: atomic {
  if
   :: (next_nrcvd < nsnt - Fnc) ->
    next_nrcvd = nrcvd + 1;
   :: next_nrcvd = nrcvd;
  fi
  /* compute */
  /* send    */
 }; goto step;
}
```

Fig. 2: Modeling the effect of faults on correct processes: Byzantine (ByzI), symmetric (SymmI), omission (OmitI), and clean crashes (CleanI).

to the impact of faulty process. For instance, in the Byzantine case, in addition to the messages sent by correct processes, a process can receive up to f messages more. This is expressed by the condition `(next_nrcvd < nsnt + F)`.

For Byzantine and symmetric faults we only model correct processes explicitly. Thus, we specify that the system has `N-F` copies of the process. Moreover, we can use Property (RelComm) to model reliable communication. Omission and crash faults, however, we model explicitly, so that we have `N` copies of processes. Without going into too much detail, the impact of faulty processes is modeled by relaxed fairness requirements: as some messages sent by these f faulty processes may not be received, this induces less strict communication fairness:

$$\mathbf{G}\,\mathbf{F}\,\neg\,[\exists i.\ rcvd_i + f < nsnt_i]$$

By similar adaptations one models, e.g., corrupted communication (e.g., due to faulty links) [31], or hybrid fault models [4] that contain different fault scenarios.

Figure 3 compares the number of states and memory consumption when modeling message passing using both solutions. We ran Spin to perform exhaustive

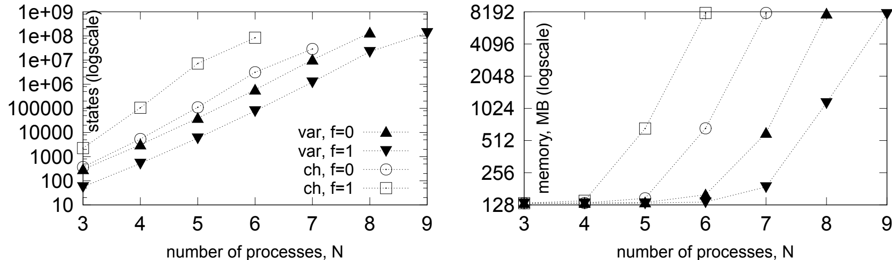

Fig. 3: Visited states (left) and memory usage (right) when modeling message passing with channels (ch) or shared variables (var). The faults are in effect only when $f > 0$. Ran with `SAFETY`, `COLLAPSE`, `COMP`, and 8GB of memory.

state enumeration on the encoding of Algorithm 1 (discussed in the next section). As one sees, the model with explicit channels and faulty processes ran out of memory on *six* processes, whereas the shared memory model did so only with *nine* processes. Moreover, the latter scales better in the presence of faults, while the former degrades with faults. This leads us to use the shared memory encoding based on *nsnt* variables.

3.3 Encoding the control flow

Recall Algorithm 1, which is written in typical pseudocode found in the distributed algorithms literature. The lines `3-8` describe one step of the algorithm. Receiving messages is implicit and performed before line `3`, and the actual sending of messages is deferred to the end, and is performed after line `8`.

We encoded the algorithm in Figure 4 using custom Promela extensions to express notions of fault-tolerant distributed algorithms. The extensions are required to express a parameterized model checking problem, and are used by our tool that implements the abstraction methods introduced in [21]. These extensions are only syntactic sugar when the parameters are fixed: `symbolic` is used to declare parameters, and `assume` is used to impose resilience conditions on them (but is ignored in explicit state model checking). Declarations `atomic <var> = all (...)` are a shorthand for declaring atomic propositions that are unfolded into conjunctions over all processes (similarly for `some`). Also we allow expressions over parameters in the argument of `active`.

In the encoding in Figure 4, the whole step is captured within an atomic block (lines 20–42). As usual for fault-tolerant algorithms, this block has three logical parts: the receive part (lines 21–24), the computation part (lines 25–32), and the sending part (lines 33–38). As we have already discussed the encoding of message passing above, it remains to discuss the control flow of the algorithm.

Control state of the algorithm. Apart from receiving and sending messages, Algorithm 1 refers to several facts about the current control state of a process:

```
1  /* parameters (instantiated with concrete values) */
2  symbolic int N, T, F;
3  /* the resilience condition */
4  assume(N > 3 * T && T >= 1 && 0 <= F && F <= T);
5  /* quantified atomic propositions */
6  atomic prec_unforg = all(STBcast:sv == V0);
7  atomic prec_corr = all(STBcast:sv == V1);
8  atomic prec_init = all(STBcast@step);
9  atomic ex_acc = some(STBcast:sv == AC);
10 atomic all_acc = all(STBcast:sv == AC);
11 atomic in_transit = some(STBcast:nrcvd < nsnt);
12
13 active[N - F] proctype STBcast() {
14   byte sv, next_sv;          /* status of the algorithm */
15   int nrcvd = 0, next_nrcvd = 0; /* nr. of echoes received */
16   if  /* initialize */
17     :: sv = V0; /* v_i = FALSE */
18     :: sv = V1; /* v_i = TRUE */
19   fi;
20 step: atomic { /* an indivisible step */
21     if /* receive one more echo (up to nsnt + F) */
22       :: (next_nrcvd < nsnt + F) -> next_nrcvd = nrcvd + 1;
23       :: next_nrcvd = nrcvd; /* or nothing */
24     fi;
25     if /* compute */
26       :: (next_nrcvd >= N - T) ->
27         next_sv = AC; /* accept_i = TRUE */
28       :: (next_nrcvd < N - T && sv == V1
29           || next_nrcvd >= T + 1) ->
30         next_sv = SE; /* remember that <echo> is sent */
31       :: else -> next_sv = sv; /* keep the status */
32     fi;
33     if /* send */
34       :: (sv == V0 || sv == V1)
35           && (next_sv == SE || next_sv == AC) ->
36         nsnt++; /* send <echo>  */
37       :: else;  /* send nothing */
38     fi;
39     /* update local variables and reset scratch variables */
40     sv = next_sv; nrcvd = next_nrcvd;
41     next_sv = 0; next_nrcvd = 0;
42     } goto step;
43 }
44 /* LTL-X formulas */
45 ltl fairness { []<>(!in_transit) } /* added to other formulas */
46 ltl relay { [](ex_acc -> <>all_acc) }
47 ltl corr { []((prec_init && prec_corr) -> <>(ex_acc)) }
48 ltl unforg { []((prec_init && prec_unforg) -> []!ex_acc) }
```

Fig. 4: Encoding of Algorithm 1 in Promela with symbolic extensions.

"sent $\langle echo \rangle$ before", "if v_i", and "$accept_i \leftarrow$ TRUE". We capture all possible control states in a finite set SV. For instance, for Algorithm 1 one can collect the set $SV = \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$, where:

- V0 corresponds to $v_i =$ FALSE, $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ is not sent.
- V1 corresponds to $v_i =$ TRUE, $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ is not sent.
- SE corresponds to the case $\mathrm{accept}_i =$ FALSE and $\langle \mathrm{echo} \rangle$ been sent. Observe that once a process has sent $\langle \mathrm{echo} \rangle$, its value of v_i does not interfere anymore with the subsequent control flow.
- AC corresponds to the case $\mathrm{accept}_i =$ TRUE and $\langle \mathrm{echo} \rangle$ been sent. A process only sets accept to TRUE if it has sent a message (or is about to do so in the current step).

Thus, the control state is captured within a single *status variable sv* over SV with the set $SV_0 = \{\mathrm{V0}, \mathrm{V1}\}$ of initial control states.

3.4 Specifications

Specifications are obtained by the broadcasting specification parts called *unforgeability*, *correctness*, and *relay* introduced in [34]:

$$\mathbf{G}\ ([\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC}]) \quad \text{(U)}$$

$$\mathbf{G}\ ([\forall i.\ sv_i = \mathrm{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = AC]) \quad \text{(C)}$$

$$\mathbf{G}\ ([\exists i.\ sv_i = \mathrm{AC}] \rightarrow \mathbf{F}\ [\forall j.\ sv_j = AC]) \quad \text{(R)}$$

Note carefully that (U) is a safety specification while (C) and (R) are liveness specifications.

4 Experiments with SPIN

Figure 4 provides the central parts of the code of our case study. For the experiments we have implemented four distributed algorithms that use threshold-guarded commands, and differ in the fault model. We have one algorithm for each of the fault models discussed. In addition, the algorithms differ in the guarded commands. The following list is ordered from the most general fault model to the most restricted one. The given resilience conditions on n and t are the ones we expected from the literature, and their tightness was confirmed by our experiments:

BYZ. tolerates t Byzantine faults if $n > 3t$,
SYMM. tolerates t symmetric (identical Byzantine [3]) faults if $n > 2t$,
OMIT. tolerates t send omission faults if $n > 2t$,
CLEAN. tolerates t clean crash faults for $n > t$.

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
				Byz				
B1	N=7,T=2,F=2	(U)	✓	3.13 sec.	74 MB	$193 \cdot 10^3$	$1 \cdot 10^6$	229
B2	N=7,T=2,F=2	(C)	✓	3.43 sec.	75 MB	$207 \cdot 10^3$	$2 \cdot 10^6$	229
B3	N=7,T=2,F=2	(R)	✓	6.3 sec.	77 MB	$290 \cdot 10^3$	$3 \cdot 10^6$	229
B4	N=7,T=3,F=2	(U)	✓	4.38 sec.	77 MB	$265 \cdot 10^3$	$2 \cdot 10^6$	233
B5	N=7,T=3,F=2	(C)	✓	4.5 sec.	77 MB	$271 \cdot 10^3$	$2 \cdot 10^6$	233
B6	N=7,T=3,F=2	(R)	✗	0.02 sec.	68 MB	$1 \cdot 10^3$	$13 \cdot 10^3$	210
				OMIT				
O1	N=5,To=2,Fo=2	(U)	✓	1.43 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O2	N=5,To=2,Fo=2	(C)	✓	1.64 sec.	69 MB	$60 \cdot 10^3$	$1 \cdot 10^6$	183
O3	N=5,To=2,Fo=2	(R)	✓	3.69 sec.	71 MB	$92 \cdot 10^3$	$2 \cdot 10^6$	183
O4	N=5,To=2,Fo=3	(U)	✓	1.39 sec.	69 MB	$51 \cdot 10^3$	$878 \cdot 10^3$	175
O5	N=5,To=2,Fo=3	(C)	✗	1.63 sec.	69 MB	$53 \cdot 10^3$	$1 \cdot 10^6$	183
O6	N=5,To=2,Fo=3	(R)	✗	0.01 sec.	68 MB	17	135	53
				SYMM				
S1	N=5,T=1,Fp=1,Fs=0	(U)	✓	0.04 sec.	68 MB	$3 \cdot 10^3$	$23 \cdot 10^3$	121
S2	N=5,T=1,Fp=1,Fs=0	(C)	✓	0.03 sec.	68 MB	$3 \cdot 10^3$	$24 \cdot 10^3$	121
S3	N=5,T=1,Fp=1,Fs=0	(R)	✓	0.08 sec.	68 MB	$5 \cdot 10^3$	$53 \cdot 10^3$	121
S4	N=5,T=3,Fp=3,Fs=1	(U)	✓	0.01 sec.	68 MB	66	267	62
S5	N=5,T=3,Fp=3,Fs=1	(C)	✗	0.01 sec.	68 MB	62	221	66
S6	N=5,T=3,Fp=3,Fs=1	(R)	✓	0.01 sec.	68 MB	62	235	62
				CLEAN				
C1	N=3,Tc=2,Fc=2,Fnc=0	(U)	✓	0.01 sec.	68 MB	668	$7 \cdot 10^3$	77
C2	N=3,Tc=2,Fc=2,Fnc=0	(C)	✓	0.01 sec.	68 MB	892	$8 \cdot 10^3$	81
C3	N=3,Tc=2,Fc=2,Fnc=0	(R)	✓	0.02 sec.	68 MB	$1 \cdot 10^3$	$17 \cdot 10^3$	81

Table 1: Summary of experiments related to [34]

In addition, we verified a folklore reliable broadcasting algorithm that tolerates crash faults, which is given, e.g., in [9]. Further, we verified a Byzantine tolerant broadcasting algorithm from [6]. For the encoding of the algorithm from [6] we were required to use two message types — opposed to the one type of the ⟨echo⟩ messages in Algorithm 1. Finally, we implemented the asynchronous condition-based consensus algorithm from [27]. We specialized it to binary consensus, which resulted in an encoding which requires four different message types.

The major goal of the experiments was to check the adequacy of our formalization. To this end, we first considered the four well-understood variants of [34], for each of which we systematically changed the parameter values. By doing so, we verify that under our modeling the different combination of parameters lead to the expected result. Table 1 and Figure 5 summarize the results of our experiments for broadcasting algorithms in the spirit of [34]. Lines B1 – B3, O1 – O3, S1 – S3, and C1 – C3 capture the cases that are within the resilience condition known for the respective algorithm, and the algorithms were verified by Spin. In Lines B4 – B6, the algorithm's parameters are chosen to achieve a goal that is known to be impossible [28], i.e., to tolerate that 3 out of 7 processes may fail.

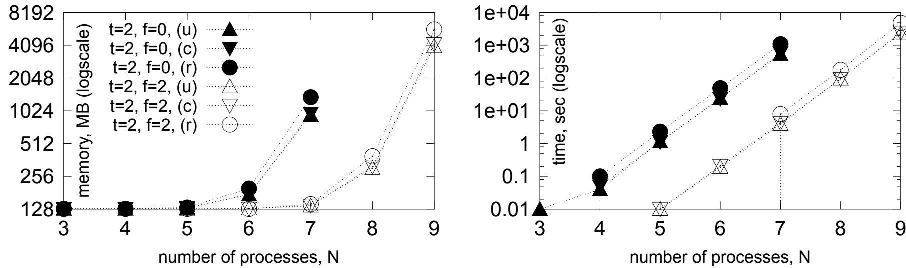

Fig. 5: Spin memory usage (left) and running time (right) for Byz.

This violates the $n > 3t$ requirement. Our experiment shows that even if only 2 faults occur in this setting, the relay specification (R) is violated. In Lines O4–O6, the algorithm is designed properly, i.e., 2 out of 5 processes may fail ($n > 2t$ in the case of omission faults). Our experiments show that this algorithm fails in the presence of 3 faulty processes, i.e., (C) and (R) are violated.

Table 2 summarizes our experiments for the algorithms in [9], [6], and [27]. The specification (F) is related to agreement and was also used in [17]. Properties (V0) and (V1) are non-triviality, that is, if all processes propose 0 (1), then 0 (1) is the only possible decision value. Property (A) is agreement and similar to (R), while Property (T) is termination, and requires that every correct process eventually decides. In all experiments the validity of the specifications was as expected from the distributed algorithms literature.

For slightly bigger systems, that is, for $n = 11$ our experiments run out of memory. This shows the need for parameterized verification of these algorithms.

5 Related Work

As fault tolerance is required to increase the reliability of systems, the verification of fault tolerance mechanisms is an important challenge. There are two classes of approaches towards fault tolerance, namely *fault detection*, and *fault masking*.

Methods in the first class follow the fault detection, isolation, and recovery (FDIR) principles: at runtime one tries to detect faults and to automatically perform counter measures. In this area, in [32] Spin was used to validate a design based on the well-known primary backup idea. Under the FDIR approach, validation techniques have also been introduced in [15,8,19].

However, it is well understood that it is not always possible to reliably detect faults; for instance, in asynchronous distributed systems it is not possible to distinguish a process that prematurely stopped from a slow process, and in synchronous systems there are cases where the border between correct and faulty behavior cannot be drawn sharply [1]. To address such issues, fault masking has been introduced. Here, one does not try to detect or isolate faults, but tries to keep those components operating consistently that are not directly hit by faults,

#	parameter values	spec	valid	Time	Mem.	Stored	Transitions	Depth
				Folklore Broadcast [9]				
F1	N=2	(U)	✓	0.01 sec.	98 MB	121	$7 \cdot 10^3$	77
F2	N=2	(R)	✓	0.01 sec.	98 MB	143	$8 \cdot 10^3$	48
F3	N=2	(F)	✓	0.01 sec.	98 MB	257	$2 \cdot 10^3$	76
F4	N=6	(U)	✓	386 sec.	670 MB	$15 \cdot 10^6$	$20 \cdot 10^6$	272
F5	N=6	(R)	✓	691 sec.	996 MB	$24 \cdot 10^6$	$370 \cdot 10^6$	272
F6	N=6	(F)	✓	1690 sec.	1819 MB	$39 \cdot 10^6$	$875 \cdot 10^6$	328
				Asynchronous Byzantine Agreement [6]				
T1	N=5,T=1,F=1	(R)	✓	131 sec.	239 MB	$4 \cdot 10^6$	$74 \cdot 10^6$	211
T2	N=5,T=1,F=2	(R)	✗	0.68 sec.	99 MB	$11 \cdot 10^3$	$465 \cdot 10^3$	187
T3	N=5,T=2,F=2	(R)	✗	0.02 sec.	99 MB	726	$9 \cdot 10^3$	264
				Condition-based consensus [27]				
S1	N=3,T=1,F=1	(V0)	✓	0.01 sec.	98 MB	$1.4 \cdot 10^3$	$7 \cdot 10^3$	115
S2	N=3,T=1,F=1	(V1)	✓	0.04 sec.	98 MB	$3 \cdot 10^3$	$18 \cdot 10^3$	128
S3	N=3,T=1,F=1	(A)	✓	0.09 sec.	98 MB	$8 \cdot 10^3$	$42 \cdot 10^3$	127
S4	N=3,T=1,F=1	(T)	✓	0.16 sec.	66 MB	$9 \cdot 10^3$	$83 \cdot 10^3$	133
S5	N=3,T=1,F=2	(V0)	✓	0.02 sec.	68 MB	1724	9835	123
S6	N=3,T=1,F=2	(V1)	✓	0.05 sec.	68 MB	3647	$23 \cdot 10^3$	136
S7	N=3,T=1,F=2	(A)	✓	0.12 sec.	68 MB	$10 \cdot 10^3$	$55 \cdot 10^3$	135
S8	N=3,T=1,F=2	(T)	✗	0.05 sec.	68 MB	$3 \cdot 10^3$	$17 \cdot 10^3$	135

Table 2: Summary of experiments with algorithms from [9,6,27]

cf. distributed agreement [28]. The fault-tolerant distributed algorithms that we consider in this paper belong to this approach.

Specific masking fault-tolerant distributed algorithms have been verified, e.g., a consensus algorithm in [36], and a clock synchronization algorithm in [35]. In [25], a bug has been found in a previously published clock synchronization algorithm that was supposed to tolerate Byzantine faults.

Formalization and verification of a class of fault-tolerant distributed algorithms have been addressed in [5]. Their formalization uses the fact that for many distributed algorithms, the order in which messages arrive is not relevant, but only how many messages are received. They provide a framework for such algorithms and show that they can be efficiently verified using partial order reduction. While in this work we consider similar message counting ideas, our formalization targets at parameterized model checking [21] rather than partial order reductions for systems of small size.

6 Conclusions

In this paper we presented a way to efficiently encode fault-tolerant threshold-guarded distributed algorithms using shared variables. We showed that our encoding scales significantly better than a straight-forward approach. With this encoding we were able to verify small system instances of a number of broadcasting algorithms [34,6,9] for diverse failure models. We could also find counter

examples in cases where we knew from theory that the given number of faults cannot be tolerated. We also verified a condition-based consensus algorithm [27].

As our mid-term goal is to verify state-of-the-art fault-tolerant distributed algorithms, there are several follow-up steps we are currently taking. In [21] we show that the encoding we described in this paper is also a basis for parameterized model checking techniques that allow us to verify the correctness of distributed algorithms for any system size. Some of the algorithms mentioned above could already be verified in this setting, while we are working on techniques to verify also the other ones. Also we are currently working on verification of the Paxos-like Byzantine consensus algorithm from [26], which is also threshold-guarded. The challenges of this algorithm are threefold. First, it consists of three different process types — proposers, accepters, learners — while the algorithms discussed in this paper are just compositions of processes of the same type. Second, to tolerate a single fault, the algorithm requires at least four proposers, six acceptors, and four learners. Our preliminary experiments show that 14 processes is a challenge for explicit state enumeration. Third, as the algorithm solves consensus, it cannot work in the asynchronous model [16], and we have to restrict the interleavings of steps, and the message delays.

References

1. Ademaj, A.: Slightly-off-specification failures in the time-triggered architecture. In: High-Level Design Validation and Test Workshop. IEEE. pp. 7–12 (2002)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with Byzantine failures and little system synchrony. In: DSN. pp. 147–155 (2006)
3. Attiya, H., Welch, J.: Distributed Computing. John Wiley & Sons, 2nd edn. (2004)
4. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theoretical Computer Science 412(40), 5602–5630 (2011)
5. Bokor, P., Kinder, J., Serafini, M., Suri, N.: Efficient model checking of fault-tolerant distributed protocols. In: DSN. pp. 73–84 (2011)
6. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32(4), 824–840 (1985)
7. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. 81, 13–31 (April 1989)
8. Bucchiarone, A., Muccini, H., Pelliccione, P.: Architecting fault-tolerant component-based systems: from requirements to testing. Electr. Notes Theor. Comput. Sci. 168, 77–90 (2007)
9. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (March 1996)
10. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
11. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In: TACAS'08/ETAPS'08. pp. 33–47. Springer (2008)
12. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: CONCUR 2004. vol. 3170, pp. 276–291 (2004)
13. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (Apr 1988)

14. Emerson, E., Namjoshi, K.: Reasoning about rings. In: POPL. pp. 85–94 (1995)
15. Feather, M.S., Fickas, S., Razermera-Mamy, N.A.: Model-checking for validation of a fault protection system. In: HASE. pp. 32–41 (2001)
16. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (Apr 1985)
17. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In: TACAS. LNCS, vol. 4963, pp. 315–331. Springer (2008)
18. Függer, M., Schmid, U.: Reconciling fault-tolerant distributed computing and systems-on-chip. Distributed Computing 24(6), 323–355 (2012)
19. Gnesi, S., Latella, D., Lenzini, G., Abbaneo, C., Amendola, A.M., Marmo, P.: A formal specification and validation of a critical system in presence of Byzantine errors. In: TACAS. LNCS, vol. 1785, pp. 535–549 (2000)
20. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
21. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Counter attack on Byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms. arXiv CoRR abs/1210.3846 (2012), (submitted to LICS)
22. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
23. Lamport, L.: On interprocess communication. part i: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
24. Lynch, N.: Distributed Algorithms. Morgan Kaufman, San Francisco, USA (1996)
25. Malekpour, M.R., Siminiceanu, R.: Comments on the "Byzantine self-stabilizing pulse synchronization". protocol: Counterexamples. Tech. rep., NASA (Feb 2006)
26. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Trans. Dep. Sec. Comp. 3(3), 202–215 (2006)
27. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the condition-based approach to solve consensus. In: DSN. pp. 541–550 (2003)
28. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J.ACM 27(2), 228–234 (April 1980)
29. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0,1,\infty)$- counter abstraction. In: CAV, LNCS, vol. 2404, pp. 93–111. Springer (2002)
30. Powell, D.: Failure mode assumptions and assumption coverage. In: FTCS-22. pp. 386–395. Boston, MA, USA (1992)
31. Santoro, N., Widmayer, P.: Time is not a healer. In: STACS. LNCS, vol. 349, pp. 304–313. Springer (1989)
32. Schneider, F., Easterbrook, S.M., Callahan, J.R., Holzmann, G.J.: Validating requirements for fault tolerant systems using model checking. In: ICRE. pp. 4–13 (1998)
33. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)
34. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2, 80–94 (1987)
35. Steiner, W., Rushby, J.M., Sorea, M., Pfeifer, H.: Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In: DSN. pp. 189–198 (2004)
36. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Computing 23(5–6), 341–358 (2011)
37. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing 20(2), 115–140 (August 2007)

APPENDIX

```
1  symbolic int N, T, Fs, Fp;
2  assume(N > 2 * T
3      && Fs <= Fp && Fp <= T);
4  ...
5  atomic in_transit
6    = some(Proc:nrcvd < nsnt + Fs);
7
8  active[N - Fp] proctype STSymm() {
9    ...
10   if /* receive */
11     :: next_nrcvd < nsnt + Fs ->
12       next_nrcvd = nrcvd + 1;
13     :: next_nrcvd = nrcvd;
14   fi;
15   if /* compute */
16     :: next_nrcvd >= T + 1 ->
17         next_sv = AC;
18     :: sv == V1 -> next_sv = SE;
19     :: else -> next_sv = sv;
20   fi;
21   /* send as in Byz... */ }
```

```
1  symbolic int N, To, Fo;
2  assume(N > 2 * To
3    && 0 <= Fo && Fo <= T);
4  ...
5  atomic in_transit
6    = some(Proc:nrcvd < nsnt - Fo);
7
8  active[N] proctype STOmit() {
9    ...
10   if /* receive */
11     :: next_nrcvd < nsnt ->
12       next_nrcvd = nrcvd + 1;
13     :: next_nrcvd = nrcvd;
14   fi;
15   if /* compute */
16     :: next_nrcvd >= T + 1 ->
17       next_sv = AC;
18     :: next_nrcvd >= 1
19         && next_nrcvd < To + 1 ->
20       next_sv = SE;
21     :: sv == V1 -> next_sv = SE;
22     :: else -> next_sv = sv;
23   fi;
24   /* send as in Byz... */ }
```

Fig. 6: Differences of SYMM and OMIT from BYZ

```
1  symbolic int N, Tc, Fc, Fnc;
2  assume(N > Tc + 1 && Fnc <= Fc && Fc <= Tc);
3  ...
4  atomic in_transit = some(Proc:nrcvd < nsnt - Fnc);
5
6  active[N] proctype STClean() {
7    ...
8    if /* receive */
9      :: next_nrcvd < nsnt - Fnc ->
10       next_nrcvd = nrcvd + 1;
11     :: next_nrcvd = nrcvd;
12   fi;
13   if /* compute */
14     :: next_nrcvd >= N - Tc -> next_sv = AC;
15     :: sv == V1 -> next_sv = SE;
16     :: else -> next_sv = sv;
17   fi;
18   /* send as in Byz... */ }
```

Fig. 7: Differences of CLEAN from BYZ

```
1  symbolic int N;
2  int nsnt, nsntF;
3  atomic in_transit
4      = some(Proc:nrcvd < nsnt);
5  ...
6  active[N] proctype STFolklore() {
7   ...
8   if /* receive */
9    :: next_nrcvd < nsnt + nsntF ->
10    next_nrcvd = nrcvd + 1;
11   :: next_nrcvd = nrcvd;
12  fi;
13  if /* compute */
14   :: sv == V1 ->
15     next_sv = AC;
16   :: sv == V1 ->
17     next_sv = CR; /* crash */
18   :: sv != AC && sv != CR && (next_nrcvd >= 1) ->
19     next_sv = AC;
20   :: sv != AC && sv != CR && next_nrcvd >= 1 ->
21     next_sv = CR;
22   :: else -> next_sv = sv;
23  fi;
24  if /* send */
25   :: (pc == V0 || pc == V1)
26    && next_pc == AC -> nsnt++;
27   :: (pc == V0 || pc == V1)
28    && (next_pc == CR) -> nsntF++;
29   :: else
30  fi;
31  ...
32 ltl fkl { [](prec_init -> <>[](!ex_acc || all_acc)) }
```

Fig. 8: Fragment of FOLKLORE BROADCAST

```
1  symbolic int N, T, F;
2  int nsnt00, nsnt01, nsnt10, nsnt11;
3  int nfaulty, init0, init1;
4  assume(N > 2 * T && F <= T);
5  atomic prec_no0 = all(Proc:pc != V0);
6  atomic prec_no1 = all(Proc:pc != V1);
7  atomic ex_acc0 = some(Proc:pc == AC0);
8  atomic ex_acc1 = some(Proc:pc == AC1);
9  atomic prec_init = ((init0 + init1) == N);
10 atomic cond_init = ((init0 > (init1 + F)) || (init1 > (init0 + F)));
11 atomic all_acc = all(Proc:pc == CR ||Proc:pc == AC0 || Proc:pc == AC1);
12 atomic in_transit00 = some(Proc:nrcvd00 < nsnt00);
13 /* similar for nrcvd01, nrcvd10, nrcvd11 */
14 active[N] proctype CondConsensus() {
15  byte pc = 0, next_pc = 0;
16  int nrcvd00, next_nrcvd00, nrcvd01, next_nrcvd01;
17  int nrcvd10, next_nrcvd10, nrcvd11, next_nrcvd11;
18  if /* INIT */
19   :: pc = V0 -> init0++;
20   :: pc = V1 -> init1++;
21  fi;
22 step: atomic {
23     if
24     :: (pc == V0 || pc == V1 || pc == P0) && (nrcvd00 < nsnt00) ->
25       next_nrcvd00 = nrcvd00 + 1;
26     :: next_nrcvd00 = nrcvd00;
27     fi; /* and similar for nrcvd01... */
28     if :: (pc == W0 || pc == W1 || pc == P1) && (nrcvd10 < nsnt10) ->
29       next_nrcvd10 = nrcvd10 + 1;
30        :: next_nrcvd10 = nrcvd10;
31     fi; /* and similar for nrcvd11... */
32     if :: pc == V0 || pc == V1 -> next_pc = P0;
33      :: (pc == P0) && ((next_nrcvd0 + next_nrcvd1) >= N - T )
34        && (next_nrcvd0 > next_nrcvd1) -> next_pc = W0;
35      :: (pc == P0) && ((next_nrcvd0 + next_nrcvd1) >= N - T )
36        && (next_nrcvd1 > next_nrcvd0) -> next_pc = W1;
37      :: pc == W0 || pc == W1 -> next_pc = P1;
38      :: (pc == P1) && (next_nrcvd0 >= ((N-1) / 2)+1) ->
39        next_pc = AC0;
40      :: (pc == P1) && (next_nrcvd1 >= ((N-1) / 2)+1) ->
41        next_pc = AC1;
42      :: nfaulty < F && pc != CR ->
43        nfaulty++; next_pc = CR;
44      :: else -> next_pc = pc;
45     fi;
46     if :: (pc == V0) && (next_pc == P0) -> nsnt00++;
47      :: (pc == V1) && (next_pc == P0) -> nsnt01++;
48      :: (pc == P0) && (next_pc == W0) ->
49       nsnt10++; next_nrcvd0 = 0; next_nrcvd1 = 0;
50      :: (pc == P1) && next_pc == W1  ->
51       nsnt11++; next_nrcvd0 = 0; next_nrcvd1 = 0;
52      :: else;
53     fi;
54     ...
55   } goto step; }
56 ltl fairness { []<>(!in_transit00 && !in_transit01
57     && !in_transit10 && !in_transit11) }
58 ltl validity0 { (([](prec_no0) && <>(prec_init)) -> []!ex_acc0) }
59 ltl agreement { [](!ex_acc0 || !ex_acc1)}
60 ltl termination { []((!prec_init || !cond_init) || (<>(all_acc))) }
```

Fig. 9: Fragment of CONDITION-BASED CONSENSUS