Model Checking Distributed Consensus

Giorgio Delzanno
DIBRIS, Università di Genova

Michele Tatarek
DIBRIS, Università di Genova

Riccardo Traverso
FBK Trento

We present a formal model of a distributed consensus algorithm in the executable specification language Promela extended with a new type of guards, called counting guards, needed to implement transitions that depend on majority voting. Our formalization exploits abstractions that follow from reduction theorems applied to the specific case-study. We apply the model checker Spin to automatically validate finite instances of the model and to extract preconditions on the size of quorums used in the election phases of the protocol.

1 Introduction

Distributed algorithms are a challenging class of case-studies for automated verification methods (see e.g. [7, 1, 18, 7, 8, 11, 19, 5]). The main difficulties come from typical assumptions taken in these algorithms such as asynchronous communication media, topology-dependent protocol rules, and messages with complex structure. In the subclass of fault tolerant distributed protocols there are additional aspects to consider that often make their validation task harder. Indeed, fault tolerant protocols are often based on dynamic leader elections in which quorums may change from one round to another. When modeling these protocols, one has to deal with a very fast growth of the search space for increasing number of processes.

Following preliminary evaluations described in [4], in this paper we apply Promela/Spin to specify and validate a fault tolerant distributed consensus protocols for asynchronous systems caled Paxos [12, 13] used in the implementation of distributed services in the Google File System [3]. Promela is a specification language for multithreaded systems with both shared memory and communication capabilities. Spin is a model checker based on state exploration for validating a finite-automata representation of Promela models. Our formal specification of Paxos is based on the formulation of the protocol given by Marzullo, Mei and Meling in [13] in which the behavior of each process is described via three separate roles (proposer, acceptor, and learner). More specifically, our contributions are as follows. (1) We give a formal Promela specification of the Paxos protocol that is modular with respect to roles, rounds, and communication media. The specification is closer to a possible implementation than sequential models with non-deterministic assignments used in other approaches, e.g., [20]. Furthermore, Promela provides a non-ambiguous executable semantics that can be tested by using the Spin simulator and model checker. (2) Via a formal analysis extracted from the correctness requirements, which are specified using auxiliary variables and assertions, we give reduction theorems that can be used to restrict the number of processes instances for some of the protocol roles, namely proposers and learners. (3) To optimize the model, we extend Promela with a new type of transition, called quorum transition, that can directly be applied to model elections via majority voting. The transition is defined on top of a special guard that counts the number of messages in a channel. The transition is embedded into Promela code using the deterministic step constructor *d_step* that can be used to atomically execute a block of Promela instructions.

Compared to [4] in which we focused our attention on the comparison of the model checkers Spin and Groove (based on graph transformation systems) on a common case-study, in this paper we focus our

Submitted to:

```
Paxos — Proposer $p$
constants:
    $A$ = set of acceptors
    $P$ = set of proposals
    $maj = \lceil (\#A+1)/2 \rceil$
init $crnd \leftarrow -1$    /* current round */
on receive $\langle Propose, val \rangle$
        from client
    /* pick fresh round in $\mathbb{N}$ */
    $crnd \leftarrow pickNextRound(crnd)$
    $P \leftarrow \emptyset$
    $tval \leftarrow val$
    send $\langle Prepare, crnd \rangle$ to $A$
on receive $\langle Promise, rnd, prnd, pval \rangle$
        with $rnd = crnd$ from acceptor $a \in A$
    $P \leftarrow P \oplus (prnd, pval)$
on event $\#P \geq maj$
    $(j, pval) = \max_{prnd}\{(prnd, pval') \in P\}$
    if $j \geq 0$ then
        $myval = pval$
    else
        $myval \leftarrow tval$
    send $\langle Accept, crnd, myval \rangle$ to $A$
```

```
Paxos — Acceptor $a$
constants:
    $L$ = set of learners
    $P$ = set of proposers
init $crnd \leftarrow -1$    /* current round */
    $prnd \leftarrow -1$    /* previous round */

on receive $\langle Prepare, rnd \rangle$ with $rnd > crnd$
        from proposer $p \in P$
    $crnd \leftarrow rnd$
    send $\langle Promise, crnd, prnd, pval \rangle$ to $p$
on receive $\langle Accept, rnd, aval \rangle$ with $rnd \geq crnd$
        from proposer $p \in P$
    $crnd \leftarrow rnd$
    $prnd \leftarrow rnd$
    $pval \leftarrow aval$
    send $\langle Learn, crnd, aval \rangle$ to $L$
```

```
Paxos — Learner $l$
constants
    $A$ = set of acceptors
    $maj = \lceil (\#A+1)/2 \rceil$
init $V \leftarrow \emptyset$
on receive $\langle Learn, rnd, lval \rangle$ from acceptor $a \in A$
    $V \leftarrow V \oplus (rnd, lval)$
on event $\exists m = (rnd, lval) : \#\{m \mid m \in V\} \geq maj$
    choose $lval$
```

Figure 1: Pseudo-code of the Paxos protocol

attention on the design choices and protocol properties that we applied to obtain the Promela specification. In our new Promela specification we also managed to reduce the state space generated by exhaustive analysis via Spin. The Groove model studied in [4] can be viewed as a declarative specification of Paxos. The state space of the Groove model is still more compact thanks to the symmetry reduction built-in in Groove. The reduction in the analysis however require a preliminary abstraction step in the modeling phase.

2 Distributed Consensus in Asynchronous Systems

In this section we briefly describe the distributed consensus problem and one possible solution due to Lamport called Paxos. The distributed consensus problem requires agreement among a number of agents, communicating via asynchronous channels, on a single data value. Some of the agents may fail, so consensus protocols are required to be fault tolerant, i.e., to reach consensus even if a subset of nodes interrupt their operations resuming their state later during the execution. Initially, each agent proposes a value to all other ones. Agents then exchange their information. A correct protocol must ensure that when a node takes the final choice, the chosen value is the same for all correct agents. The asynchronous communication assumption, i.e., messages can be delayed arbitrarily, represents the worst case scenario for possible solutions to the problem. Indeed, Fisher, Lynch and Patterson have shown that, under the above mentioned assumption, solving consensus is impossible [9].

In [12] Lamport proposed a (possibly non terminating) algorithm, called Paxos, addressing this problem. Paxos is based on the metaphor of a part-time parliament, in which part-time legislators need to keep consistent records of their passing laws. However the description was hard to understand, so Lamport later provided a simpler description of the protocol in [13] in terms of three separate agent roles: proposers that can propose values for consensus, acceptors that accept a value among those proposed, and learners that learn the chosen value.

The pseudo-code of the algorithm is shown in Fig 1. The code is obtained as a slight variation of the formulation of the protocol given by Marzullo, Mei and Meling in [16][1]. In a first step the proposer selects a fresh round identifier and broadcasts it to a (subset of) acceptors. It then collects votes for that round number from acceptors. Acceptor's replies, called *promises*, contain the round number, and a pair consisting of the last round and value that they accepted in previous rounds (with the same or a different proposer). When the proposer checks that majority is reached, it selects a value to submit again to the acceptors. For the selection of the value, the proposer inspects every promise received in the current round and selects the value with the highest round. It then submits the current round and the chosen value to the acceptors. Acceptors wait for proposals of round identifiers but accept only those that are higher than the last one they have seen so far. If the received round is fresh, acceptors answer with a promise not to accept proposals with smaller round numbers. Since messages might arrive out-of-order, even different proposals with increasing rounds of the same proposer might arrive with arbitrary order (this justifies the need of the promise message). Acceptors also wait for accept messages: in that case local information about the current round are updated and, if the round is fresh, the accepted pair $(round, value)$ is forwarded to the learner. A learner simply collects votes on pairs $(round, value)$ sent by acceptors and waits to detect majority for one of them.

The protocol is guaranteed to reach consensus with n acceptors up to $f = \lfloor (n-1)/2 \rfloor$ simultaneous failures, but only if learners have enough time to take a decision (i.e. to detect a majority). If proposers indefinitely inject new proposals the protocol may diverge. Under the above mentioned condition on f and in absence of byzantine faults, correctness can be formulated as follows.

Property 1 *When a value is chosen by a learner, then no other value has been chosen/accepted in previous rounds of the protocol.*

This means that, whenever a value is chosen by the learner, any successive voting always select the same value (even with larger round identifiers), i.e., the algorithm stabilizes w.r.t. the value components of tuples sent to the learner. To ensure this property, in the first part of the protocol the proposer does not immediately send its proposal for the value but only its round number. The proposer selects a value only after having acquired knowledge on the values accepted in previous rounds. Among them the choice is done by taking the value of the highest round. Its own proposal comes into play only if all values received by acceptors are undefined (equal to -1). Other safety requirements are that chosen values are among those proposed and that chosen values (by the whole system) are the same as those learned by learners.

[1]The third rule of a proposer is split in two cases depending on the value of j.

3 A Formal Model in Promela

In this section we present a formal specification of Paxos given in Promela, a specification language for multithreaded and distributed programs. Promela thread templates are defined via the `proctype` construct. The body of the template (a sequence of commands) defines the behavior of its instances. The language has a C-like syntax for shared and local data structures and instructions. Guarded commands are used to model non-deterministic choices in the body of process templates. For instance, the command `if` $:: g_1 \rightarrow c_1;\ \dots\ :: g_n \rightarrow c_n;$ `fi` specifies a non-deterministic conditional: only one command c_i among those for which the guard g_i is enabled is selected for execution. The guarded command `do` $:: g_1 \rightarrow c_1;\ \dots\ :: g_n \rightarrow c_n;$ `od` is the iterative version of the conditional. Data structures include basic data types (byte, bool, int) as well as arrays and structures. Furthermore, channels can be used for the specification of inter-process communications. For instance, `chan` $c[MAX]$, where MAX is a constant, defines a channel with at most MAX places. A message $\langle m_1,\dots,m_n \rangle$ is sent by using the command $c!m_1,\dots,m_n$, where c is a channel and $m_1,\dots,m_n$ are expressions whose current values are sent on the channel. Reception is defined via the capability $c?x_1,\dots,x_n$, where c is a channel and $x_1,\dots,x_n$ are variables used to store the data from the incoming message. Channels can be viewed as global arrays. The selector `?` provides FIFO access, whereas `??` provides random access. To restrict reception to a given pattern, it is possible to put either a constant value in a reception or an expression like $eval(x)$ that evaluates to the current value of x. We describe next our Promela model for Paxos.

A round is defined via a proposer process running in parallel with the other processes (other proposers, acceptors and learners). The proposer `proctype` takes in input two parameters: a unique round identifier (a number) and the proposed value. Round identifiers must be unique for the algorithm to work. Roles can be viewed as threads definitions within the same process situated in a location. Asynchronous communication is modeled via global channels with random receive actions to model out-of-order message delivery. We do not model message duplication explicitly but we assume that messages persist in a channel (i.e. they can be read multiple times). We forbid multiple occurrences of the same message in a channel in order to obtain upper bounds on a channel size. Persistence of messages allows us to model the majority voting using test applied directly on channels. Rounds can be viewed as time-stamps. Indeed they are required in order to fix an order on incoming `prepare` and `accept` messages. A special learner process is used to observe the results of the handshaking between proposers and acceptors, and to choose pairs `round,value`. The algorithm guarantees that once a value is chosen, such a choice remains stable when other (old/new) proposals are processed by the agents. The learner keeps a set of counters indexed on rounds to check for majority on a value.

We discuss next the Promela specification in full detail. First of all, we use the following constants:

```
#define ACCEPTORS 3
#define PROPOSERS 5
#define MAJ (ACCEPTORS/2+1)// majority
#define MAX (ACCEPTORS*PROPOSERS)
```

MAJ defines defines the size of quorums. By changing MAJ we can infer preconditions on the number of faulty processes. MAX defines an upper bound on the size of channels.

Using a thread-like style, we consider four shared data structures that represent communication channels. They correspond to different phases of the protocol. Their signature is defined as follows.

```
typedef mex{
        byte rnd;
        short prnd;
        short pval;
}
```

```
chan prepare = [MAX] of { byte, byte };
chan accept  = [MAX] of { byte, byte, short };
chan promise = [MAX] of { mex };
chan learn   = [MAX] of { short, short, short };
```

The message signature is defined as follows. The proposer of round *r* sends messages `prepare(i,r)` and `accept(i,r,v)` to acceptor *i*. Acceptor *i* sends the message `promise(r,hr,hval)` to the proposer of round *r* and `learn(i,r,v)` to the learner. All channels are treated as multiset of messages using the random receive operations `??`.

The protocol makes use of broadcast communication (from proposers to acceptors). We implement a derived broadcast primitive using Promela macro definition via the inline declaration. Specifically, we add the `baccept` and `bprepare` primitives defined as follows, where *i* is an integer index local to a process proctype.

```
inline baccept(round,v){
    for(i:1.. ACCEPTORS){
       accept!!i,round,v;
    }
    i=0;
}

inline bprepare(round){
    for(i:1.. ACCEPTORS){
      prepare!!i,round;
    }
    i=0;
}
```

In the above listed definitions for each process identifiers we insert the message specified in the parameter resp. in the `accept` and `prepare` channels. The typedef *mex* will be used later to inspect the content of a channel using a for-loop (Promela allows this kind of operations on channels in which messages have a predefined type).

In the above macros we always use ordered channel insertion, via the !! operator, in order to maintain a canonical representation of channel contents, i.e., to identify state-vectors that differ only in the order in which the same set of messages are listed in the channels. Finally, at the end of each for-loop we reset the auxiliary variable *i*, in order to end the broadcast message in a state that is independent from the size of the channel. Again, this is a way to avoid the creation of unnecessary states.

We now move to the formal specification of the three protocol roles. They can be specified in a modular way by using three separate *proctype* declarations.

3.1 Proposer Role

A single round of a proposer is defined via the `proctype` in Fig. 2. The round is identified by a unique value passed as a parameter to the `proctype`. The other parameter, `myval`, is the proposed text/value of the proposer. To generate several proposals, it is necessary to create several instances of the proposer `proctype` running in parallel. Together with the use of unordered channels, our definition models arbitrary delays among the considered set of proposals (old proposals can overtake new ones). We assume here that proposers are instantiated with distinct round values.

Inside the proctype we use different local variables. In particular, `count` is used to count votes (i.e. promise messages) and to check if a majority has been reached in the current round. `hr` and `hval` are local variables used to store resp. the max round identifier seen in `promise` messages and the associated value. The subprotocol starts with a `bprepare` invocation embedded into a deterministic step. This way, the proposal message is atomically sent to all processes, i.e., a copy of the message is added to the channel `propose` channel with the identifier of each acceptor.

The broadcast is followed by a non-deterministic loop consisting of a single rule that models a quorum transition as described in the following paragraph.

Counting Guards and Quorum Transitions

A possible way to model the behavior of a proposer is to specify a reception rule for promise messages to keep track in a local counter of the number of promises that contain the round of the considered proposer. A second rule can be used to trigger the event associated to the detection of a majority of promises for the considered round. Counting up messages using counters on reception of promises/learn messages has a major drawback, i.e., the introduction of a number of intermediate states proportional to the steps needed to reach a majority (every time we receive a message we increment a counter). To eliminate these auxiliary states, we need a new type of guards that are able to count the occurrences of a given message in a channel. We will refer to them as *counting guards*. When used in a conditional statement they would allow us to atomically check if a quorum has been reached for a specific candidate message, i.e., to implement atomic *quorum transitions*. Counting guards are not supported directly by Promela/Spin. However they can be implemented by using the `d_step` (deterministic step) construct. This construct transforms a block of transitions into a single not interruptible step. A counting guard on channel *ch* and message *m* is based on the following idea. We consider the *ch* channel as a circular queue. We then perform a for-loop as many times as the current length of the channel. At each iteration we read message *m* by using the FIFO read operation ? and reinsert it at the end of the channel so as to inspect its content without destroying it. We then increment an occurrence counter *count* if the message *m* contains the proposed round identifier. The other fields of the messages are inspected as well in order to search for the value of the promise containing the maximal round identifier. At the end of the for-loop we can test the counter to fire the second part of the transition in which we test if the majority has been reached. An accept message is broadcasted to the acceptors when majority is detected.

We remark that the counting guard is done in a single deterministic step, i.e., it does not introduce any intermediate states (all auxiliary variables are reset at the end of the loop). Furthermore, we still consider possible interleaving between different quorums since acceptors are free to reply to proposers in any order. The resulting Promela code is shown in Fig. 2.

3.2 Acceptor Role

An acceptor is defined via the proctype of Fig. 3. We use the following local variables: `crnd` contains the current round, `prnd`, `pval` contain resp. the maximal round identifier and the associated value seen in previous accept messages. Valid values must be greater or equal than zero. The template consists of a non-deterministic loop with two options. In the first option on receipt of a `prepare` message containing a round identifier larger than the current one, the acceptor updates `crnd` and answers to the proposer with a promise message. The promise contains the values `crnd,prnd,pval` used by the proposed to select a value. In the second option, on receipt of an `accept` message containing a round identifier larger than the current one, the acceptor updates `crnd,prnd,pval` and sends a notification to the learner containing a proposal `prnd,pval` for the second votation.

3.3 Learner Role

The learner role is defined by the proctype of Fig. 4. A learner keeps track, in the array `mcount`, of the number of received proposals values in each round (the counter `mcount[r]` is associated to round `r`). We assume that in each round there is a unique accepted value selected by the proposer (no byzantine faults). This is the reason why we can just count the number of received message in a given round.

3.4 Initial Configuration

For a fixed number of processes the initial configuration of the system is defined using Promela as in the following `init` command:

```
init
{ atomic{
   run proposer(1,1); run proposer(2,2);
   run acceptor(0); run acceptor(1); run acceptor(2);
   run learner(); }; }
```

The `atomic` construct enforce atomic execution of the initial creation of process instances. In this example we consider three possible proposals that are sent in arbitrary order (proposers run in parallel). Their round identifiers are 1 and 2 and the associated values are 1 and 2, respectively. The system has three acceptors with identifiers $0-2$ and a single learner.

4 Formal Analysis

We consider here the property of Def. 1, which requires that learners always choose the same values. Before discussing how to encode the property, we make some preliminary observations. Since messages are duplicated and seen by any process, we observe that we can restrict our model in order to consider a single learner process that is always reacting to incoming `learn` messages. The parallel execution of several learners is then modeled via several rounds of a single learner.

Theorem 1 *The safety property of Def. 1 holds for the model with multiple learners if and only if there exist no execution that violates the safety assertion in the model with a single instance of the* `learner` *proctype.*

Proof Assume that the property is violated in the model containing a single instance of `learner`. This implies that the instance of the learner performs k iterations in which the learned value is always the same and an additional iteration in which the value is distinct from the previous one. Since the learner only observes incoming messages, we can run the same execution of the protocol with distinct instances of the learner process. To make the proposition stronger let us assume that each learner learns a single value.

We observe that we just need two distinct instances of the learner process to get a violation of the safety requirement. We can then run the same instance of the protocol with the single learner. Since communication is asynchronous, we can assume that the messages needed in the first $k-1$ are delayed arbitrarily, and just consider the pairs (r,v) and (r',v') learned in steps k and $k+1$, respectively. The two instances of the learner will learn such pairs and the safety requirement will be violated in the resulting system.

We now assume that safety is violated in the model with multiple learners. This implies that there exist two distinct pairs (r,v) and (r',v') with $v \neq v'$ learned by two distinct learners (again we assume that each learner learns a single pair). Again we can run the same execution of the protocol, delay all messages not involving such pairs, and let the `learner` process execute two iterations learning them. Clearly, the safety assertion will be violated after the votation in the second iteration. □

Another important observation is that, since we do not consider byzantine faults, it is not possible that two distinct values are proposed in the same round. Thus, in order to detect a majority we keep an array of counters (one for each round). The counter for round r is incremented when a message for that round is observed in the `learn` channel. Under this assumption we need to show that, once the learner has detected a majority vote for a given value, then the chosen value cannot change anymore. We define then the `active proctype` for the single learner process of Fig. 5. The idea is to add an auxiliary variable `lastvalue` in which we store the last learned value. Every time a new majority is detected the learner compares the corresponding value to `lastvalue`. An alarm is raised if the two values are not the same. The alarm is modeled via the assertion $assert(false)$ (or $assert(lastval == v)$). Since we consider a single learner process that abstracts the behavior of a collection of learners, we tag the proctype as `active`, i.e., the corresponding process will start together with those specified in the initial configuration (we remove `run learner()` from `run`).

Apart from the reduction of the number of learners, we can also reduce the number of proposers. Indeed, to expose violations we just need two proposers proposing distinct values. This property is formalized in the following statement.

Theorem 2 *If for a given value of the parameter MAJ the safety property of Def. 1 holds for two proposers (with distinct values), then it holds for any number of proposers.*

Proof Assume a given $k \geq 0$. By contraposition, we show that if there exists an execution of $k > 2$ proposers that violates the assertion in the `single_learner` code, then there exists an execution with 2 proposers (distinct rounds and values) that violates the assertion. We consider the first round r that violates the assertion, i.e., such that the pair (r, v') obtains a majority observed by the learner for a value v' distinct from the value v stored in `last_val` (i.e. the value learned in all previous observed votations). We now have to show that we can construct another execution in which we just need two rounds r_v and $r_{v'}$, namely the rounds in which values v and v' have been proposed. Let us assume that $r_v < r_{v'}$. The other case is also possible, e.g., for two independent executions involving distinct majorities. For simplicity we focus on the former case. To prove that we can define an execution involving rounds $r_{v'}$ and r_v that violates the property, we need to reason on the history of the protocol phases that produces the necessary majorities. We start by inspecting a node n that sent the learn message for the pair (r, v'). More specifically, we consider the status of its local variables before the accept message containing (r, v'):

- if they are both undefined, i.e., n did not participate to previous handshakes, n can be reused for an execution involving only $r_{v'}$ and v';
- if they contain round r_1 and value v, with $r_1 < r$, then we have to show that the history of node n is independent from votations involving round r for the value v'.
  - If n has never voted for value v' (i.e. the value v' has never been stored in its local state), then we can inspect the history until the first vote done by n for the value v. If the associated round is r_v, we can simply build an execution in which node n sends the learned message (r_v, v). If the round number is different from r_v, then the previous local state has undefined values for the variables (since we assume that this is the first votation). Thus, we can build an execution in which node n receives the proposal (r_v, v) directly from the proposer r_v.
  - If n has voted for value v' in a round $r_1 < r$ but the value has not reached a majority, i.e., the learner never observed a majority for (r_1, v'), then the vote of node n plays no role for the

Prop	Acc	Max	Maj	Time (sec)	App	States	Unsafe	OutOfMem
2	2	4	2	<0,01		266		
2	3	6	2	0.06		20,529		
2	4	8	3	0.83		200,391		
2	5	10	3	129		19,364,052		
2	6	12	4	_		_		X
3	2	6	2	0,01		4,986		
3	3	9	2	9.62		2,292,025		
3	4	12	3	388		53,945,064		
3	5	15	3	_		_		X
4	2	8	2	0.37		89,755		
4	3	12	2	_		_		X
2	5	10	2	0,218	X	65,142	X	
3	4	12	2	1,42	X	305,837	X	
4	3	12	1	0,187	X	40,755	X	

Table 1: Experimental results with finite protocol instances.

election of v'. Thus, we can apply the same reasoning as in the previous point, in order to move back to a state from which we can extract an execution involving only r_v and v.

$\square$

5 Experimental Analysis

In our experiments we consider the number of proposer and acceptors as distinct parameters both ranging from 2 to 8, the other dependent variable being the maximum size of channels (communication is asynchronous) set as follows: $MAJ = |ACCEPTORS| * |PROPOSERS|$, and the the maximum number of faulty processes allowed in the system. We use Spin as back-end solver to explore the values of the parameters and to extrapolate the minimal constraints for ensuring the correctness of the protocol. In our experiments we considered the parameters in Table 1. The first and second columns show the increasing number of proposers and acceptors considered in the experiments. As their number increases we can notice the typical exponential growth in the number of states, intrinsic of the model checking problem. When our pc revealed to be not sufficiently powerful to exhaustively look for the correctness of the system in every single possible state and an out-of-memory error was generated, an approximate search option was used (App stands for -DBITSTATE option given for compilation process along with 3 bits per state and default hash-table size chosen for the verification phase). In that case Spin bitstate hashing turned out to be useful when quorums where not enough to reach a majority and a counterexample was given almost in any case. The correct approximate case instead has little meaning because, even increasing the internal hash table size and using more bits to represent a state, the hash factor remains well below a good states coverage target. In any case, the abstractions used to reduce the state-space proved to be successful to analyse a few processes and to tackle the complexity of distributed system verification. When approximate search was not needed we compressed the state descriptors using the appropriate Spin option to fully verify more effectively. Spin was run on a 6GB machine using a presetting of 4GB for the state vector. All experiments have been done with a PC equipped with an Intel i7 quad-core with hyper-threading enabled by default. Our models are available here: `http://www.disi.unige.it/person/DelzannoG/PAXOS/`.

6 Related Work

Formal specification in temporal and first order logic, TLA, of Paxos and its variants together with automatically checked proofs are given in [15, 14]. Differently from the TLA-based approach, our analysis is based on a software model checking approach based on automata combined with abstractions and heuristics (e.g. counting guards). The efficiency of model checking message passing protocols is studied in [2], which compares different semantics for asynchronous communication (e.g. one in which special delivery events are used to move messages from output to input buffers and another in which messages are directly placed in input buffers). The authors conclude the article with some results obtained by applying the discussed message passing models to Paxos, with two proposers and up to four acceptors. Depending on the specific model and property being considered, the state space varies from about 5×10^4 states up to 1.7×10^6. An approach for bounded model checking asynchronous round-based consensus protocols is proposed in [20], where the authors exploit invariants and over-approximations to check agreement and termination properties on two variants of Paxos. In [10], the authors focus on fault tolerant distributed algorithms and notice how such algorithms often rely on threshold guards to take transition steps. They propose an approach specific to this class of protocols. While [10] considers more types of faulty processes than we do, like Byzantine failures, the modeled protocols are simpler under several aspects (e.g, processes are all executing the same code, and they exchange messages from a small, finite alphabet of message types). Reduction theorems and abstractions for other examples of distributed systems (e.g. mutual exclusione protocols) are considered in [6, 17].

7 Conclusions

In this paper we presented a formal model for the Paxos algorithm given in terms of finite state automata described in the high level language Promela. Reasoning on the size of the automata and reachability states produced by Spin, we managed to define different types of optimizations, e.g., based on a new type of guards that atomically inspect the content of a channel (counting guards). We also consider two reduction properties that can be used to limit some of the unbounded dimension of the specification (number of proposers and learners). Finally, we use the optimizations and heuristics built-in in Spin to scale up the analysis and to validate the protocol preconditions by generating counterexamples for quorum sizes that do not respect the correctness requirements. Our experiments show that the combination of abstractions, heuristics, and model checking can be used to analyze challenging examples of distributed algorithms even for large number of process instances (beyond the limit considered in previous attempts like [2, 20] that do not exploit overapproximations).

References

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Invisible safety of distributed protocols. In *ICALP (2)*, pages 528–539, 2006.

[2] P. Bokor, M. Serafini, N., and Suri. On efficient models for model checking message-passing distributed protocols. In *Formal Techniques for Distributed Systems*, pages 216–223. Springer, 2010.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06*, pages 335–350, 2006.

[4] G. Delzanno, A. Rensink, and R. Traverso. Graph- versus vector-based analysis of a consensus protocol. In *GRAPHITE*, 2014.

[5] G. Delzanno and R. Traverso. Specification and validation of link reversal routing via graph transformations. In *SPIN*, page 238249, 2013.

[6] E. Allen Emerson and Kedar S. Namjoshi. On reasoning about rings. *IJFCS*, 14(4):527–550, 2003.

[7] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. Automated analysis of aodv using uppaal. In *TACAS*, pages 173–187, 2012.

[8] A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the lmac protocol for wireless sensor networks. In *IFM*, pages 253–272, 2007.

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

[10] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *Model Checking Software*, pages 209–226. Springer, 2013.

[11] S. Joshi and B. König. Applying the graph minor theorem to the verification of graph transformation systems. In *CAV*, pages 214–226, 2008.

[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(3):133–169, 1998.

[13] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32, 4(121):51–58, 2001.

[14] L. Lamport. Byzantizing paxos by refinement. In *25th International Symposium: DISC 2011*, pages 211–224, 2011.

[15] L.Lamport and E. Gafni. Disk paxos. *Distributed Computing*, pages 1–20, 2003.

[16] K. Marzullo, A. Mei, and H. Meling. A simpler proof for paxos and fast paxos. Course notes, 2013.

[17] K. S. Namjoshi and R. J. Trefler. Uncovering symmetries in irregular process networks. In *VMCAI*, pages 496–514, 2013.

[18] M. Saksena, O. Wibling, and B. Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In *TACAS*, pages 18–32, 2008.

[19] A. Singh, C. R. Ramakrishnan, and S. A. Smolka. Query-based model checking of ad hoc network protocols. In *CONCUR*, pages 603–619, 2009.

[20] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In *Distributed Computing*, pages 466–480. Springer, 2008.

```
proctype proposer(short crnd; short myval) {
 short aux, hr = -1, hv = -1;
 short rnd;
 short prnd,pval;
 byte count=0,i=0;
 mex pr;

 d_step{
 bprepare(crnd);
 }
 do
 :: atomic{
  d_step{
   do
   i=0;
   :: i < len(promise) ->
     promise?pr;
     promise!pr;
     if
       ::pr.rnd==crnd->
        count++;
        if
          ::pr.prnd>hr->
              hr=pr.prnd;
              hv=pr.pval;
        ::else
       fi;
       ::else
     fi;
     i++;
   :: else ->
     pr.prnd=0;
     pr.pval=0;
     pr.rnd=0;
     i=0;
     break;
   od;
   }
   if
    ::count>=MAJ ->
      aux=(hr<0 ->myval : hv);
      baccept(crnd,aux);
      break;
    ::else
   fi;
   hv= -1;
   hr = -1;
   count =0;
   aux=0;
  }
  od
}
```

Figure 2: Proposer template.

```
proctype acceptor(int id) {
 short crnd = -1, prnd = -1, pval = -1;
 short aval,rnd;
 do
   ::d_step {
     prepare??eval(id),rnd ->
     if
      :: (rnd>crnd)  ->
       crnd=rnd;
       promise!!crnd,prnd,pval;
      :: else
    fi;
    rnd = 0 /* reset */
   }
   :: d_step {
     accept??eval(id),rnd,aval ->
       if
        :: (rnd>=crnd) ->
         crnd=rnd;
         prnd=rnd;
         pval=aval;
         learn!!id,crnd,aval;
        :: else
      fi;
      rnd = 0; aval = 0 /* reset */
    }
 od
}
```

Figure 3: Acceptor template.

```
proctype learner() {
  short id, rnd, lval;
  byte mcount[PROPOSERS];
  do ::
  d_step {
    learn??id,rnd,lval ->
    if  :: mcount[rnd-1] < MAJ -> mcount[rnd-1]++;
        :: mcount[rnd-1] >= MAJ -> printf("%d_%d_chosen!",rnd,lval);
    fi;
    id = 0; rnd = 0; lval = 0 /* reset */
    }
  od
}
```

Figure 4: Learner process template.

```
active proctype learner() {
 short lastval = -1, id, rnd, lval;
 byte mcount[PROPOSERS];

 do
 ::
 d_step {
  learn??id,rnd,lval ->
    if
     :: mcount[rnd-1] < MAJ ->
          mcount[rnd-1]++;
     :: else
     fi;
     if
     :: mcount[rnd-1] >= MAJ    ->
       if :: (lastval >= 0 && lastval != lval) ->
              assert(false);
          :: (lastval == -1) -> lastval = lval;
          :: else
       fi
     :: else
     fi;
     id = 0; rnd = 0; lval = 0
  }
 od
}
```

Figure 5: Safety property embedded in the single learner.