Graph- versus Vector-Based Analysis of a Consensus Protocol

Giorgio Delzanno
DIBRIS
Università di Genova (Italy)

Arend Rensink
Department of Computer Science
University of Twente (Netherlands)

Riccardo Traverso
DIBRIS
Università di Genova (Italy)
FBK-irst
Trento (Italy)

The Paxos distributed consensus algorithm is a challenging case-study for standard, vector-based model checking techniques. Due to asynchronous communication, exhaustive analysis may generate very large state spaces already for small model instances. In this paper, we show the advantages of graph transformation as an alternative modelling technique. We model Paxos in a rich declarative transformation language, featuring (among other things) nested quantifiers, and we validate our model using the GROOVE model checker, a graph-based tool that exploits isomorphism as a natural way to prune the state space via symmetry reductions. We compare the results with those obtained by the standard model checker SPIN on the basis of a vector-based encoding of the algorithm.

1 Introduction

Automated validation of distributed algorithms like routing and consensus protocols is a challenging task for state-of-the-art model checkers [4, 5, 14, 13, 8, 3, 1]. These protocols often depend on the current network topology and operate under asynchronous communication assumptions. These two features are a frequent cause of state space explosion. To attack this problem, in [3] we have proposed to apply an unconventional model checking approach based on Graph Transformation Systems (GTS). Graph grammars provide a declarative language to specify updates (of structure and labels) in a graph-based representation of a dynamic system. Apart from providing an alternative description of the problem that may in itself be interesting, the formalisation of states as graphs rather than the more conventional vectors of data values opens the way towards additional techniques for state space reduction.

In this paper we focus our attention on a graph-based declarative specification of the Paxos distributed consensus algorithm. The consensus problem requires agreement among a number of agents for a single data value. Some of the agents may fail, so consensus protocols must be fault tolerant. Initially, each agent proposes a value to all other ones. Agents can then exchange information. A correct protocol must ensure that when a node takes the final choice, the chosen value is the same for all correct agents. It is assumed here that messages can be delayed arbitrarily. A subset of processes can crash anytime and restore their state (keeping their local information) after an arbitrary delay.

Fisher, Lynch and Patterson have shown that, under the these assumptions, deterministically solving consensus is impossible [6]. In [10] 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. Because the description proved hard to understand, Lamport later provided a simpler description of the protocol in [9]. This is the version on which we base the models in this paper.

Our declarative specification is based on graph transformation rules with symbolic conditions on node attributes, negative application conditions and node quantification, as provided in the GROOVE framework [7]. Salient features of the specification are:

Dragan Bošnački, Stefan Edelkamp, Alberto Lluch Lafuente
& Anton Wijs (Eds.): 3rd Workshop on
GRAPH Inspection and Traversal Engineering (GRAPHITE 2014)
EPTCS 159, 2014, pp. 44–57, doi:10.4204/EPTCS.159.5

- We use node occurrences as abstractions of proposed values and process identifiers. This causes symmetries to show up as graph isomorphism between states.
- We use a Linda-like model for asynchronous communication, in which message broadcasts are represented by special nodes linked to their senders, without associated buffers or channels. This avoids state differences due to irrelevant message orderings.
- Our rules encapsulate a lot of functionality within a single (atomic) transformation step, and thus avoid intermediate states during evaluation.

We compare the resulting models computed by GROOVE to those generated by SPIN from a specification of the same protocol in PROMELA. The comparison shows that the choices listed above manage to keep the graph-based state space size to a fraction of that of a more traditional vector-based specification, enabling the analysis of larger problem instances despite the inherent complexity of graph transformation.

Our method can be seen as an attempt of combining declarative reasoning and efficient search methods for this class of protocols. Furthermore, it represents an alternative to standard model checking frameworks based on (unstructured) symbolic representations, e.g., BDDs.

2 The Paxos Consensus Algorithm

The description of Paxos in [9] distinguishes three separate agent roles: *proposers* that can propose values for consensus, *acceptors* that accept a value among those proposed, and *learners* that learn the accepted values and eventually choose one of them. We present the protocol on the basis of a pseudo-code description from the lecture notes [11].

In a first step, the proposer selects a fresh round identifier and broadcasts it to all acceptors, in a message called *Prepare*. It then collects votes for that round number from live acceptors. Acceptors' replies, called *Promise*s, contain the round number and a pair consisting of the last round and value that they promised in previous rounds (with the same or a different proposer). Rounds and values are initialized to the default of -1, and only change upon *Accept* messages. When the proposer checks that a majority is reached, it selects a value to submit again to the acceptors. For the selection of this value, the proposer inspects every *Promise* received in the current round and selects the value with the highest non-default round; if it did not receive a non-default value, it uses its own initial proposal (*myval*). It then submits the current round and the chosen value to the acceptors, in a message called *Accept*.

Acceptors wait for proposals (i.e., *Prepare* messages) of round identifiers but consider only those that are *fresh*, in the sense of being 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 *Prepare*s with increasing rounds of the same proposer might arrive in 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 is updated and, if the round is fresh, the accepted pair $(rnd, aval)$ is forwarded to the learner, in a message called *Learn*.

A learner collects votes (*Accept* messages) on pairs $(rnd, lval)$ sent by acceptors and waits to detect a majority for one of them. When a majority is detected, the *lval* component is chosen.

The pseudo-code of the algorithm, based on [11], is given in Fig. 1. Majority is defined as *maj*. The pseudo-code includes a special *Propose* message that corresponds to an external command sent to the node in order to inject a new proposal (a value) into the whole system. In the proposer code, *pickNextRound* must return a fresh value (w.r.t. all processes) for the next round, and *pick* must return the value associated to a tuple with highest round number. We use $\oplus$ to denote multiset union (which is required to count multiple occurrences of the same pair).

Paxos — Proposer p

constants
A = set of acceptors
$maj = \lceil (\#A+1)/2 \rceil$
init $crnd \leftarrow -1$ /* *current round* */
on $\langle Propose, val \rangle$
/* *pick fresh round* */
$crnd \leftarrow pickNextRound(crnd)$
$myval \leftarrow val$
$P \leftarrow \emptyset$
send $\langle Prepare, crnd \rangle$ **to** A
on $\langle Promise, rnd, prnd, pval \rangle$ with $rnd = crnd$
from acceptor a
$P \leftarrow P \oplus (prnd, pval)$
on event $\#P \geq maj$
$j = \max\{prnd \mid (prnd, pval) \in P\}$
if $j \geq 0$ **then**
$V = \{pval \mid (j, pval) \in P\}$
/* *pick value with largest prnd* */
$myval \leftarrow pick(V)$
send $\langle Accept, crnd, myval \rangle$ **to** A

Paxos — Acceptor a

constants L = set of learners
init $crnd \leftarrow -1$ /* *current round* */
$prnd \leftarrow -1$ /* *previous round* */
$pval \leftarrow -1$ /* *previous value* */
on $\langle Prepare, rnd \rangle$ with $rnd > crnd$
from proposer p
$crnd \leftarrow rnd$
send $\langle Promise, crnd, prnd, pval \rangle$ **to** p
on $\langle Accept, rnd, aval \rangle$ with $rnd \geq crnd$
from proposer 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 $\langle Learn, rnd, lval \rangle$ **from** acceptor 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

The protocol is guaranteed to reach consensus for $maj \geq \lceil (\#A+1)/2 \rceil$, where $\#A$ denotes the size of the set A of correct acceptors, and only if acceptors and learners have enough time to take a decision (i.e., to detect a majority). If proposers indefinitely inject new proposals, the protocol may diverge. In this paper we will concentrate on the following limited notion of correctness:

Definition 1 (safety) *The protocol is correct if, when a value is chosen by a learner, it has been proposed by a proposer, and no other value has been chosen by any learner in previous rounds of the protocol.*

This means that, whenever a value is chosen by a learner, any successive choices always select the same value (possibly with larger round identifiers); i.e., the algorithms stabilizes w.r.t. the value components of tuples sent to the learners.

Simplifying assumptions. In both the GROOVE and the SPIN model presented in this paper, we have made the following simplifying assumptions about the protocol:

- Proposers never send more than one *Prepare* message. This does not restrict the protocol for the purpose of the correctness criterion in Def. 1 (the effect of multiple *Prepare* messages may be mimicked by increasing the number of proposers) but it causes the protocol to always terminate.
- There is only a single learner. This cannot affect the correctness of the protocol either, as all learners have access to exactly the same information and hence are bound to have the same behaviour. In other words, any error in a scenario with multiple learners must necessarily occur already with a single learner.

3 A Graph-Based Model of the Paxos Algorithm

In the graph-based model, the global states of the protocol are captured by single graphs. Each such graph is typed according to the type graph in Fig 2.

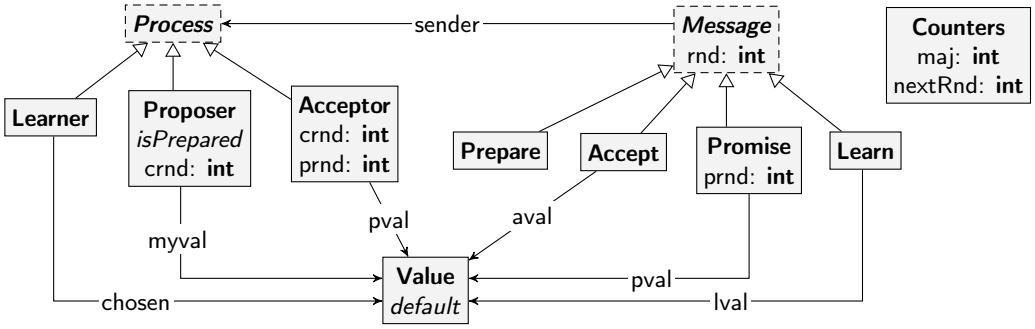

Figure 2: Type graph of the Paxos protocol

As the type graph shows, there are two abstract types, **Process** and **Message**: each **Message** has a round number rnd during which it was sent, and a sender (which is a **Process**). In addition there is a type **Counters**, which will always have a singular instance that serves as a container for the global variables maj (the bound considered to be a majority) and nextRnd (an auxiliary variable used to dispense initial round numbers). There are three types of **Process** and four types of **Message**, corresponding to the roles and messages of the protocol. The arrows and attributes correspond to the local fields and variables discussed in Sect. 2. In addition, the following may be noted:

- Processes and messages have no explicit identities. This is important in order to ensure that symmetrical states give rise to isomorphic graphs.
- **Proposer** instances have a flag isPrepared, which will be set when a proposer has sent a **Prepare** message and is ready to receive **Promise**s.
- The values proposed and chosen by the protocol are not represented as integers but as nodes of type **Value**. The default flag on **Value** will be used to distinguished the default value with which all acceptors are initialized (-1 in the pseudocode of Fig. 1).

Fig. 3 shows an initial configuration with three proposers, four acceptors and a majority bound of 2. The protocol is expected to be incorrect in this case, as the majority does not exceed half of the acceptors.

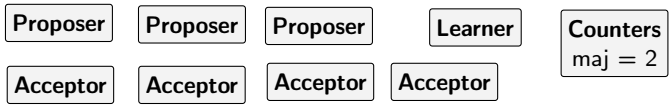

Figure 3: Example initial configuration of the Paxos protocol

Initialization. The dynamics of the protocol are captured by transformation rules. In addition, the model is equipped with a control program to schedule the rules. Fig. 4 shows the main control loop as well as the initialization rule.

The control loop specifies that the rule initValue is to be invoked, followed by a perpetual choice between proposer, acceptor and learner actions, as specified by the functions proposer() etc. (see below).

The rule deserves clarification. The $\forall$-quantifiers cause all nodes connected with dashed @-labelled arrows to be matched as often as possible. The fat, gray **Value** nodes (green in a coloured view) are

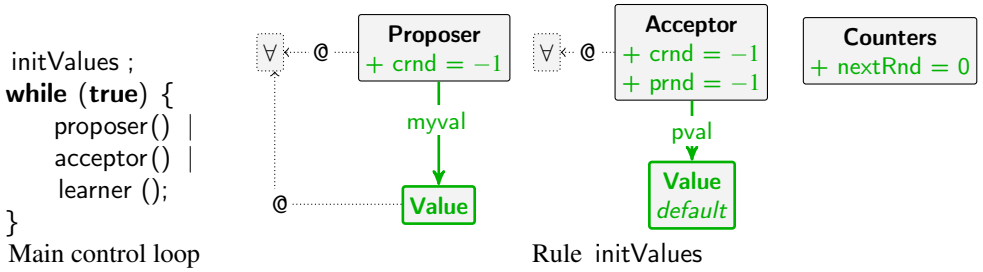

Figure 4: Top-level control and initialization rule

created as a result of the rule, as are the +-prefixed attributes in the **Proposer**, **Acceptor** and **Counters** nodes. For instance, applying the rule changes the graph of Fig. 3 into Fig. 5.

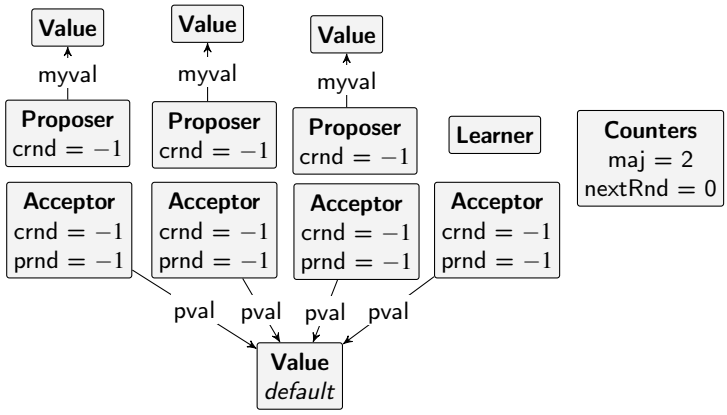

Figure 5: Initial configuration of Fig. 3 after application of initValues

Proposers. Fig. 6 shows the control function and the rules that encompass the functionality of proposers, as specified by the pseudocode in Fig. 1. The control function proposer() specifies a non-deterministic choice between the rule onPropose on the one hand, corresponding to the "**on** *Propose*" clause of Fig. 1, and a sequence of onPromise, changeMyval and sendAccept on the other (where **try** causes changeMyval to be applied only if possible), corresponding to the "**on** *Promise*" and "**on event**" clauses. The parameter prop ensures that the rules are applied to the same proposer instance.

- Rule onPropose specifies the update of the proposer's crnd attribute and the creation of a **Prepare** message, under the condition that crnd < 0. Moreover, the isPrepared flag is set.
- Rule onPromise tests if the number of **Promise** messages with this proposer's round number exceeds the majority bound. Note that prom.count, where prom refers to the ∀-quantifier, stands for the number of matches of the @-connected subgraph — in this case, just the **Promise** node. The isPrepared flag is a precondition for this rule and is at the same time deleted, making sure that each proposer can execute this event only once.
  The adornment in the top left of the **Proposer** indicates that this node is a rule parameter. When the rule is applied, the value of this parameter is bound to the prop-variable in the control program.
- Rule changeMyval adjusts the myval field to the pval of the promise with the highest prnd value, but only if that is not the default value. (The dashed **Promise**-node — red in a coloured view— with the !=-labelled edge to the top right **Promise** specifies that there is *no* promise with a *higher* prnd.)

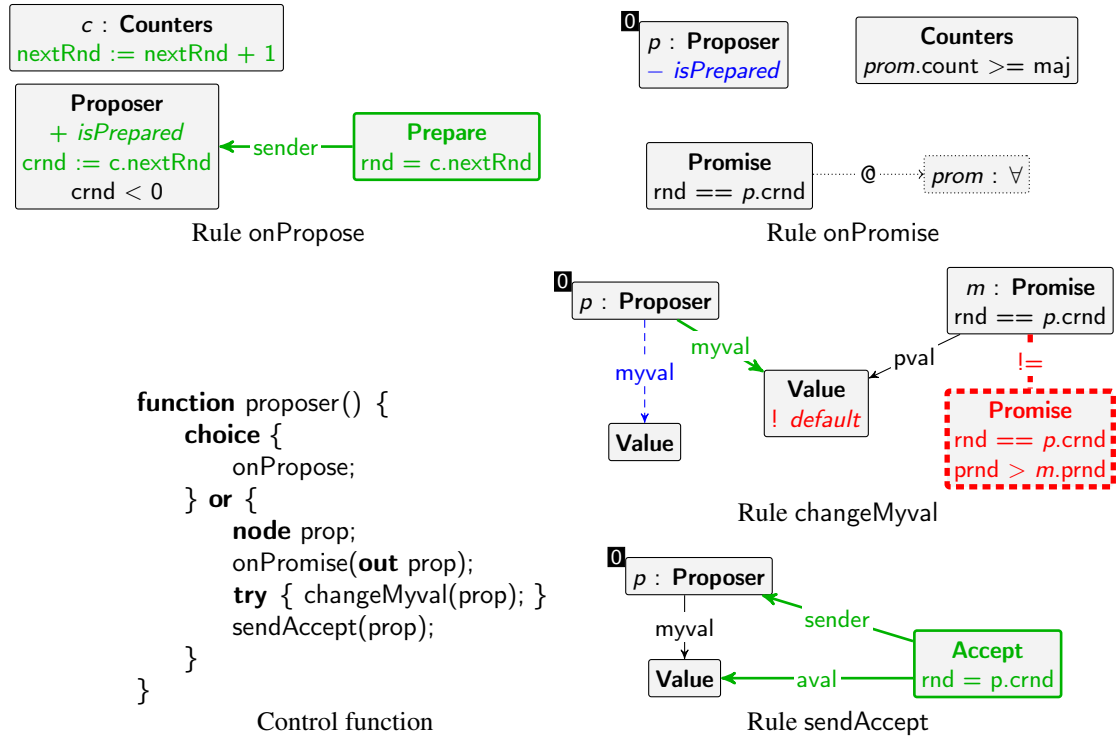

Figure 6: Proposer behaviour

- Finally, rule sendAccept specifies that an **Accept** message is sent. Due to the scheduling in the control program, this only occurs after changeMyval has had a chance to be applied.

Acceptors and learners. Fig. 6 shows the control function and rules for the acceptor and learner roles. The control functions merely specify a choice between rules, which in turn capture the corresponding part of the pseudocode of Fig. 1. We will discuss the rules.

- Rule onPrepare creates a **Promise** message upon discovery of a **Prepare** with the right rnd.
- Rule onAccept creates a **Learn** message upon discovery of an **Accept** with the right rnd. The dashed (red) **Learn**-node is a negative condition ensuring that the rule is applicable at most once for any given **Acceptor** and **Accept**.
- Rule onLearn counts the number of **Learn** messages with identical rnd and lval fields, in the same way as onPromise of Fig. 6, and chooses the corresponding **Value** if the count has reached the majority bound. Note that there is nothing to prevent this rule from being applied more than once; however, the same value will be chosen every time.

Correctness. Based on the combination of control and rules presented above, GROOVE can generate (and optionally visualise) the state space. Moreover, for the purpose of actually validating a model against a set of requirements, GROOVE has built-in LTL and CTL model checkers. However, for the problem at hand, model checking is overkill, as we just want to check safety as defined in Def. 1. To achieve this, it suffices to try and find graphs that are *unsafe*. If that attempt fails, the protocol is correct for the initial configuration. The negated safety property is captured by the rules in Fig. 8. Note that neither of these rules actually modifies the graph.

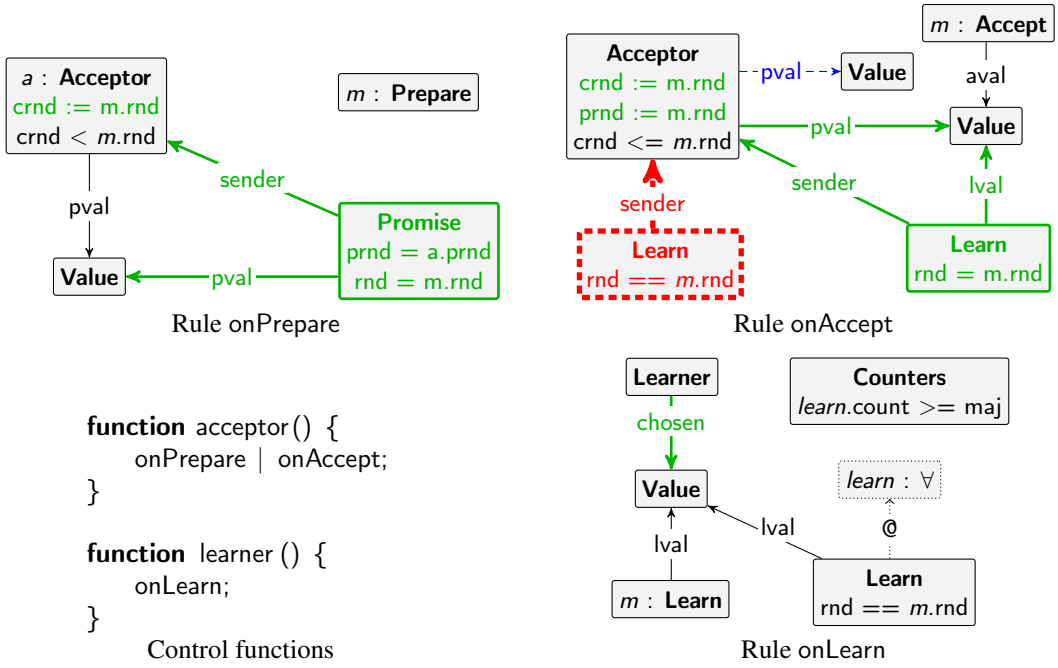

Figure 7: Acceptor and learner behaviour

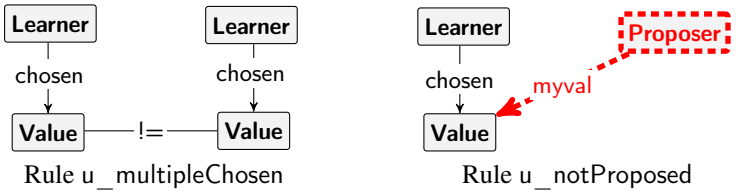

Figure 8: Safety rules encoding the negation of the property in Def. 1

- Rule u_multipleChosen tests whether two **Learner**s have chosen distinct **Value**s. The distinctness is explicitly required by the !=-edge. Due to the fact that rules may be matched non-injectively, this rule is also applicable to a graph in which a *single* **Learner** has chosen two distinct **Value**s.
- Rule u_notProposed tests whether a **Learner** has chosen a **Value** that has not been proposed by any **Proposer**.

GROOVE supports a range of different exploration strategies. For this particular case it can do a depth-first search that halts as soon as a graph is found that satisfies the propositional formula u_multipleChosen || u_notProposed. If no such state exists, this strategy will cause the entire state space to be searched, potentially encompassing (many) millions of states.

4 A Vector-Based Model of the Paxos Algorithm

We now present a formal specification of Paxos in PROMELA, the input language of the model checker SPIN. A PROMELA specification consists of a number of processes communicating through shared channels. The processes are obtained by instantiating so-called **proctype** templates. We call this type of model *vector-based* because of the way the states are encoded during state space exploration, namely as

vectors of values (representing local variables and channel contents).

Initialization. For Paxos, we define three **proctype**s, corresponding to the roles of the protocol. For instance, the initial configuration shown for GROOVE in Fig 3 is specified in PROMELA as

```
init { atomic {
  run proposer(1,1); run proposer(2,2); run proposer(3,3);
  run acceptor(0); run acceptor(1); run acceptor(2); run acceptor(3);
  run learner();
} }
```

We will discuss the template definitions and their parameters below. The channels and majority bound are defined as follows:

```
#define MAJ 3   /* majority bound */
#define MAX 10  /* maximum channel capacity */
chan prepare = [MAX] of { byte, byte };          /* prop to acc:  id, crnd        */
chan accept  = [MAX] of { byte, byte, short };   /* prop to acc:  id, crnd, myval */
chan promise = [MAX] of { byte, short, short };  /* acc to prop:  crnd, prnd, pval */
chan learn   = [MAX] of { short, short, short }; /* acc to learn: id, crnd, aval  */
```

Every channel conveys messages of one of the four types in the protocol. Any process can access any of the channels for sending or receiving. For the purpose of this protocol we always use !! to insert messages in channels in lexicographic order and ?? to select the first matching message from them; this effectively turns the channels into multisets of messages while keeping channels in canonical form (ordered lists).

Templates. Fig. 9 shows the proposer and acceptor templates. Proposers are identified by a unique value passed in as the first parameter, crnd. The other parameter, myval, is the proposed value. The template starts with a bprepare invocation, which causes a broadcast of prepare messages. This is followed by a non-deterministic **do**–**od** loop which is exited once an accept message has been broadcast. In this loop, in an atomic step, first the messages in the promise buffer with the right rnd parameter are counted by iterating over all the messages (using a temporary artificial message with non-existent round number as marker); then, if the count exceeds the majority, the accept is broadcast.

The acceptor template of Fig. 9 is very similar to the pseudocode of Fig. 1. The id parameter is required to address individual acceptors in prepare and accept messages. Finally, the learner template is defined in Fig. 10. The counters mcount[rnd] keep track of the number of received learn messages associated to round rnd.

Modelling choices. In order to keep the state space size as small as possible, we have taken the following measures in the PROMELA model:

- All communication uses the ordered send and unordered receive operators !! and ??.
- Sequences of statements are wrapped in **d_step** and **atomic** blocks wherever possible. Their execution is thus represented by a single transition, not interrupted by other processes (unless a statement blocks inside an **atomic** statement).
- The proposer uses a "quorum transition" scheme for atomically counting the relevant messages, while avoiding a state change in the promise buffer.
- Local variables are reset to their initial values when they are no longer needed.
- Message count variables are no longer increased after they reach the majority bound.

```
proctype proposer(int crnd; short myval) {
  short rnd, prnd, pval, hr=−1, hv=−1;
  byte count;
  bprepare(crnd);
  do :: atomic {
        d_step {
          promise !9,0,0;
          do :: promise?rnd,prnd,pval −>
                if :: rnd == 9 −> break;
                   :: rnd == crnd −>
                      count++;
                      if :: prnd > hr −>
                            hr = prnd; hv = pval
                         :: else fi ;
                   :: else fi ;
                promise!rnd,prnd,pval
          od }
        if :: count >= MAJ −>
              baccept(crnd, hv<0 −> myval : hv);
              break
           :: else fi ;
        prnd=0; pval=0; hv=−1; hr=−1; count=0;
  } od
}
```

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

```
inline baccept(rnd,val) {
  accept !!0, rnd, val; accept !!1, rnd, val; accept !!2, rnd, val; accept !!3, rnd, val
}
inline bprepare(rnd) {
  prepare !!0, rnd; prepare !!1, rnd; prepare !!2, rnd; prepare !!3, rnd
}
```

Figure 9: Proposer and acceptor templates, with inlined broadcast primitives.

5 Analysis

GROOVE experimental results. Table 1 shows the outcome of running GROOVE on the graph-based model with different start states. These results were obtained using an Intel i7–2600 (64 bits) CPU at 3.4 GHz under Windows 7, running Java 6 with 8 GB of memory.

Each row reports the state count and time used for state space exploration with a given number of proposers (first column) and acceptors (second column), for a majority bound that is too low (columns 3–5), respectively just high enough (columns 6–8) for the protocol to be correct. In all cases where the bound is too low, a violation of the safety properties (Def. 1) was found well before the full state space was explored; in contrast, with the bound high enough, no violations were found and hence all reachable states were exhaustively enumerated. The number of states in the second case is typically several orders of magnitude higher than in the first case. Every next step not reported in the table (a higher number of proposers for the same number of acceptors, or vice versa) causes the full state space exploration to run out of memory.

Figure 11 shows two graphs derived from the results in Table 1. The first graph shows how the state space size (for full exploration) scales with the number of acceptors, for 2, 3 and 4 proposers. We call

```
active proctype learner() {
  short lastval = −1, id, rnd, lval;
  byte mcount[MAX];
  do :: d_step {
          learn??id,rnd,lval −>
          if :: mcount[rnd] < MAJ −> mcount[rnd]++ :: else fi;
          if :: mcount[rnd] >= MAJ −>
                if :: lastval >= 0 && lastval != lval −> assert(false)
                   :: lastval == −1 −> lastval = lval
                   :: else fi
             :: else fi;
          id = 0; rnd = 0; lval = 0 /* reset */
  } od
}
```

Figure 10: Learner process template.

		Low majority (w. iso)			High majority (with iso)			High majority (no iso)		
Proc	**Acc**	**Maj**	**States**	**Time (ms)**	**Maj**	**States**	**Time (ms)**	**Maj**	**States**	**Reduct**
2	2	1	45	110	2	78	160	2	224	65.18%
2	3	1	55	150	2	757	790	2	6,882	89.00%
2	4	2	174	360	3	1,279	1,460	3	31,256	95.91%
2	5	2	222	550	3	9,729	6,970	3	1,178,114	99.17%
2	6	3	723	1,820	4	15,783	11,521	4	–	–
2	7	3	911	3,100	4	92,289	69,316	4	–	–
2	8	4	2,574	4,960	5	143,376	131,028	5	–	–
2	9	4	3,154	4,982	5	665,564	844,890	5	–	–
2	10	5	7,729	10,581	6	992,044	1,529,461	6	–	–
2	11	5	9,233	29,963	6	3,820,671	7,698,559	6	–	–
2	12	6	20,192	40,804	7	5,491,406	14,794,452	7	–	–
3	2	1	51	120	2	677	581	2	6,674	89.86%
3	3	1	62	160	2	32,899	7,032	2	1,079,582	96.95%
3	4	2	610	842	3	98,330	25,983	3	–	–
3	5	2	407	900	3	3,880,277	6,681,550	3	–	–
3	6	3	1,481	3,411	4	12,247,549	8,771,175	4	–	–
4	2	1	58	140	2	6,082	2,020	2	260,910	97.67%
4	3	1	246	350	2	1,523,338	940,095	2	–	–
4	4	2	1,258	1,710	3	9,337,923	4,523,411	3	–	–
5	2	1	66	150	2	55,420	9,131	2	12,743,315	99.57%
6	2	1	75	170	2	506,370	77,888	2	–	–
7	2	1	85	190	2	4,607,455	1,154,561	2	–	–

Table 1: GROOVE model checking results, with and without isomorphism reduction

attention to the following phenomena:

- The vertical axis has a logarithmic scale; clearly the state space grows exponentially with the number of acceptors. Moreover, the growth much is accelerated for larger numbers of proposers, from less then 1 order of magnitude for each next acceptor (2 proposers) to around 2 orders of magnitude (4 proposers)
- The increase has a slight sawtooth shape, due to the fact that the majority bound does not increase

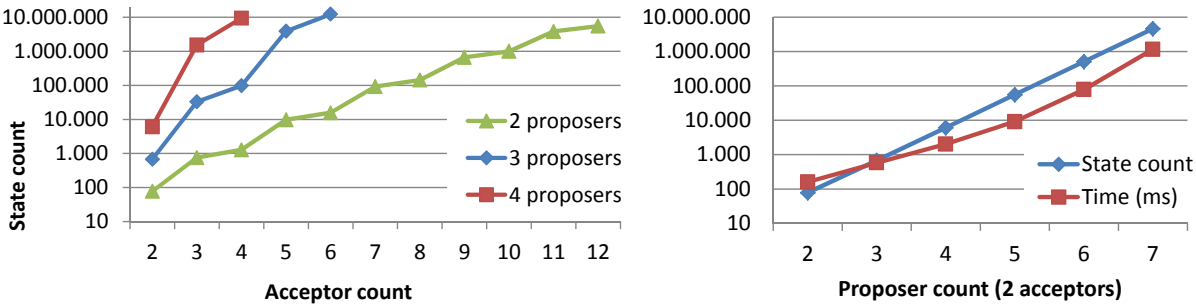

Figure 11: Graphs for the results of Table 1

with every next acceptor, but with every *second* additional acceptor.

The second graph shows the state space size and running time for an increasing number of proposers and a fixed number of 2 acceptors. The following can be remarked:

- The increase is again exponential; in fact, the exponent is larger than for the case of 2 proposers and increasing number of acceptors. This can be explained by the fact that, in contrast to the acceptors, the proposers essentially do *not* engender symmetry, since they are assigned identities through the *crnd* attribute.
- The time required by the analysis essentially keeps step with the state space size, being slightly worse for smaller problem sizes due to initialization effects and for larger problem sizes due to garbage collection.

Finally, we want to draw attention to the tool performance, in terms of number of states generated per second. This fluctuates for different problem sizes, from under 400 to over 6000. The symmetries in the pool of acceptors cause the analysis to be slower for p proposers and a acceptors than for a proposers and p acceptors, as the isomorphism checking necessary to detect these symmetries is computationally expensive (see [12] for an extensive discussion of the principles behind the symmetry reduction of GROOVE). In fact, profiling shows that for $(p,a) = (2,11)$, which has the lowest rate of generated states/s, over 75% of the total computation time is spent in isomorphism checking.

Isomorphism reduction. The last two columns of Table 1 also show the reduction due to isomorphism checking, for those problem instances for which GROOVE was able to compute the state space without reduction. Clearly, the gain is enormous, and easily justifies the time spent in computing isomorphism. The same effect was reported, more extensively, in [2].

SPIN experimental results. To compare graph- and vector-based representations, we have applied the Spin model checker to the Paxos Promela model presented above. The experiments in Table 2 are obtained on an i7 processor with 6GB of RAM running jSpin (spin 6.2.5 May 2013) under Windows 7 with default option for memory management. Without use of underapproximated search, with 2 proposers Spin is capable of finding violations up to 5 acceptors, whereas it can prove safety up to 4 acceptors. With 3 proposers, violations cannot be detected with more than 3 acceptors, and safety cannot be proved for more than 2 acceptors. A similar result is obtained for 4 proposers. By applying underapproximation heuristics like bit-hashing, Spin can deal with larger configurations while still detecting violations with low majorities, as shown by the $*$-marked columns in Table 2.

		Low majority					High majority					
P	**A**	**Maj**	**States**	**States***	**Time**	**Time***	**Maj**	**States**	**States***	**hf**	**Time**	**Time***
2	2	1	338	338	6	5	2	620	62 0		6	8
2	3	1	1,671	1,671	22	22	2	29,352	29,352		277	272
2	4	2	40,000	40,000	3,339	3,270	3	269,419	269,419		3,5	3,510
2	5	2	226,306	387,982	4,470	4,480	3	–	20,808,669	6,5	–	229,000
2	6	3	4,374,568	4,374,568	118,000	62,900	4	–	60,221,751	2,2	–	72,950,000
3	2	1	3,533	3,533	31	26	2	19,536	19,536		197	114
3	3	1	37,868	37,868	643	362	2	4,998,934	4,997,226	26	71,700	43,700
3	4	2	2,712,145	2,711,977	68,400	37,000	3	–	59,296,034	2,3	–	706,000
4	2	1	38,628	38,628	9,050	5,230	2	617,129	617,129		8,500	5,600
4	3	1	870,245	870,245	21,600	12,100	2	–	26,245,402	5,1	–	339,000
4	4	2	–	65,115,933	–	1,290,000	3	–	70,809,409	1,9	–	1,350,000

Table 2: SPIN model checking results (time is in ms): – indicates an out of memory error; * indicates results obtained via bit-hashing: hf indicates the hash factor (it indicates a high level of coverage when $>$100).

6 Conclusions

In this paper we have presented a declarative model of the Paxos consensus algorithm obtained via graph transformation rules as those used as input language for the Groove simulator and model checker. The use of extended graph transformation rules, e.g., with negative conditions and nested quantification, allows to naturally compile pseudo-code in a declarative specification in Groove. Furthermore, when compared to analysis with more traditional verification tools like Spin, experimental results with the Groove model checker show an impressive reduction of the state-space obtained via symmetry reductions based on graph isomorphism. These reductions fully exploit the underlying graph-representation of the configurations in which a key point is to use anonymous nodes to denote values and identifiers (without need of introducing integers or other enumerative types). For two proposers, Figure 12 shows the difference, in logarithmic scale, between Groove with iso-check and Spin with bitstate hashing.

The considered case-study and experimental comparison show that, well engineered graph-based search engines like Groove can compete with vector-based enumerative engines on non trivial protocol case-studies like Paxos. Comparisons with symbolic model checkers on this kind of protocols and other examples of distributed algorithms seems an interesting research direction to understand the limit of declarative model checker tools like Groove.

References

[1] Péter Bokor, Marco Serafini & Neeraj Suri (2010): *On Efficient Models for Model Checking Message-Passing Distributed Protocols*. *Lecture Notes in Computer Science*, pp. 216–223. Available at `http://dx.doi.org/10.1007/978-3-642-13464-7_17`.

[2] Pepijn Crouzen, Jaco van de Pol & Arend Rensink (2008): *Applying formal methods to gossiping networks with mCRL and groove*. *SIGMETRICS Perform. Eval. Rev.* 36(3), p. 7. Available at `http://dx.doi.org/10.1145/1481506.1481510`.

[3] Giorgio Delzanno & Riccardo Traverso (2013): *Specification and Validation of Link Reversal Routing via Graph Transformations*. *Lecture Notes in Computer Science*, pp. 160–177. Available at `http://dx.doi.org/10.1007/978-3-642-39176-7_11`.

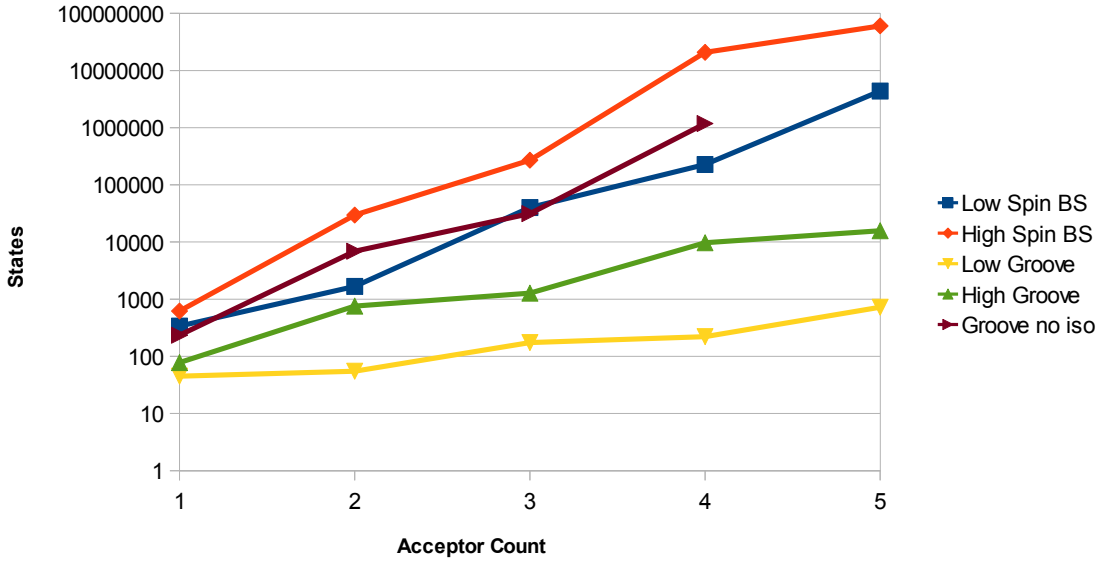

Figure 12: GROOVE versus SPIN state space sizes, 2 proposers

[4] Ansgar Fehnker, Rob van Glabbeek, Peter Höfner, Annabelle McIver, Marius Portmann & Wee Lum Tan (2012): *Automated Analysis of AODV Using UPPAAL*. *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 173–187. Available at `http://dx.doi.org/10.1007/978-3-642-28756-5_13`.

[5] Ansgar Fehnker, Lodewijk Hoesel & Angelika Mader (2007): *Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks*. *Lecture Notes in Computer Science*, pp. 253–272. Available at `http://dx.doi.org/10.1007/978-3-540-73210-5_14`.

[6] Michael J. Fischer, Nancy A. Lynch & Michael S. Paterson (1983): *Impossibility of distributed consensus with one faulty process*. *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems - PODS'83*. Available at `http://dx.doi.org/10.1145/588058.588060`.

[7] Amir Hossein Ghamarian, Maarten Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and analysis using GROOVE*. *Int J Softw Tools Technol Transfer* 14(1), pp. 15–40. Available at `http://dx.doi.org/10.1007/s10009-011-0186-x`.

[8] I. Konnov, H. Veith & J. Widder (2012): *Who is afraid of Model Checking Distributed Algorithms?* Unpublished contribution to: CAV Workshop $(EC)^2$. Available at `http://forsyte.at/wp-content/uploads/2012/07/ec2-konnov.pdf`.

[9] L. Lamport (2001): *Paxos Made Simple*. *ACM SIGACT News (Distributed Computing Column)* 32(4), pp. 51–58, doi:10.1145/568425.568433.

[10] Leslie Lamport (1998): *The part-time parliament*. *TOCS* 16(2), pp. 133–169. Available at `http://dx.doi.org/10.1145/279227.279229`.

[11] K. Marzullo, A. Mei & H. Meling (2013): *A Simpler Proof for Paxos and Fast Paxos*. Course notes.

[12] A. Rensink (2007): *Isomorphism Checking in* GROOVE. In A. Zündorf & D. Varró, editors: *Graph-Based Tools (GraBaTs)*, *Electronic Communications of the EASST* 1, European Association of Software Science and Technology.

[13] Mayank Saksena, Oskar Wibling & Bengt Jonsson (2008): *Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols*. *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 18–32. Available at `http://dx.doi.org/10.1007/978-3-540-78800-3_3`.

[14] Anu Singh, C. R. Ramakrishnan & Scott A. Smolka (2009): *Query-Based Model Checking of Ad Hoc Network Protocols*. *CONCUR 2009 - Concurrency Theory*, pp. 603–619. Available at `http://dx.doi.org/10.1007/978-3-642-04081-8_40`.