Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM*

Marta Kwiatkowska[1], Gethin Norman[1], and Roberto Segala[2]

[1] School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK
{M.Z.Kwiatkowska,G.Norman}@cs.bham.ac.uk
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy
segala@cs.unibo.it

Abstract. We consider the *randomized consensus* protocol of Aspnes and Herlihy for achieving agreement among N asynchronous processes that communicate via read/write shared registers. The algorithm guarantees termination in the presence of stopping failures within polynomial expected time. Processes proceed through possibly *unboundedly many* rounds; at each round, they read the status of all other processes and attempt to agree. Each attempt involves a *distributed random walk*: when processes disagree, a shared coin-flipping protocol is used to decide their next preferred value. Achieving polynomial expected time depends on the probability that all processes draw the same value being above an appropriate bound. For the non-probabilistic part of the algorithm, we use the proof assistant Cadence SMV to prove validity and agreement *for all N* and *for all rounds*. The coin-flipping protocol is verified using the probabilistic model checker PRISM. For a finite number of processes (up to 10) we automatically calculate the minimum probability of the processes drawing the same value. The correctness of the full protocol follows from the separately proved properties. This is the first time a complex randomized distributed algorithm has been mechanically verified.

1 Introduction

Randomization in the form of coin-flipping is a tool increasingly often used as a symmetry breaker in distributed algorithms, for example, to solve leader election or consensus problems. Such algorithms are inevitably difficult to analyse, and hence appropriate methods of automating their correctness proofs are called for. Furthermore, the use of random choices means that certain properties become probabilistic, and thus cannot be handled by conventional model checking tools.

We consider the *randomized consensus* protocol due to Aspnes and Herlihy [1] for achieving agreement among N asynchronous processes that communicate via read/write shared registers, which guarantees termination in the presence of

* Supported in part by EPSRC grants GR/M04617 and GR/M13046.

stopping failures in polynomial expected time. Processes proceed through possibly *unboundedly many* rounds; at each round, they read the status of all other processes and attempt to agree. Each agreement attempt involves a *distributed random walk* (a Markov decision process, i.e. a combination of nondeterministic and probabilistic choices): when processes disagree, a shared coin-flipping protocol is used to decide their next preferred value. Achieving polynomial expected time depends in an essential way on ensuring that the *probability* that all non-failed processes draw the same value being above an appropriate bound.

One possible approach to analyse this algorithm is to verify it using a probabilistic model checker such as PRISM [6]. However, there are a number of problems with this approach. Firstly, the model is infinite. Secondly, even when we restrict to a finite model by fixing the number of processes and rounds, the resulting models are very large: 9×10^6 states for the simpler (exponential expected time) protocol with 3 processes and 4 rounds. Thirdly, many of the requirements are non-probabilistic, and can be discharged with a conventional model checker. Therefore, we adopt a different approach, introduced by Pogosyants, Segala and Lynch [15]: we separate the algorithm into two communicating components, one non-probabilistic (an asynchronous parallel composition of processes which periodically request the outcome of a coin protocol) and the other probabilistic (a coin-flipping protocol shared by the processes). For the non-probabilistic part we use the proof assistant Cadence SMV[1], which enables us to verify the non-probabilistic requirements *for all* N and *for all rounds* by applying the reasoning introduced in [14]. The shared coin-flipping protocol is verified using the probabilistic model checker PRISM. For a finite number of processes (up to 10) we are able to mechanically calculate the minimum probability of the processes drawing the same value, as opposed to a lower bound established analytically in [1] using random walk theory. The correctness of the full protocol (for the finite configurations mentioned above) follows from the separately proved properties.

This is the first time a complex randomized distributed algorithm has been mechanically verified. Our proof structure is similar to the non-mechanical proof of [15], but the proof techniques differ substantially. Although we did not find any errors, the techniques introduced here are applicable more generally, for example, to analyse leader election [10] and Byzantine agreement [5].

Related work: The protocol discussed in this paper was originally proposed in [1], then further analysed in [15]. Distributed algorithms verified with Cadence SMV for any number of processes include the bakery algorithm [14]. We know of two other probabilistic model checkers, ProbVerus [2] and $\mathsf{E}\vdash\mathsf{MC}^2$ [9] (neither of which supports nondeterminism that is essential here).

2 The Protocol

Consensus problems arise in many distributed applications, for example, when it is necessary to agree whether to commit or abort a transaction in a distributed

[1] `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`

database. A *distributed consensus protocol* is an algorithm for ensuring that a collection of distributed processes, which start with some initial value supplied by their environment, eventually terminate agreeing on the same value. Typical requirements for such a protocol are:

Validity: If a process decides on a value, then it is the initial value of a process.
Agreement: Any two processes that decide must decide on the same value.
Termination: All processes eventually decide.

A number of solutions to the consensus problem exist (see [11] for overview). There are several complications, due to the type of model (synchronous or asynchronous) and the type of failure tolerated by the algorithm. If the processes can exhibit *stopping failures* then the **Termination** requirement is too strong and must be replaced by **Wait-free termination:** All initialized and non-failed processes eventually decide. Unfortunately, the fundamental impossibility result of [7] demonstrates that there is *no deterministic* algorithm for achieving wait-free agreement in the asynchronous distributed model with communication via shared read/write variables even in the presence of one stopping failure[2]. One solution is to use randomization, which necessitates a weaker termination guarantee:

Probabilistic wait-free termination: *With probability 1*, all initialized and non-failed processes eventually decide.

The algorithm we consider is due to Aspnes & Herlihy [1]. It is a complex algorithm using a sophisticated shared coin-flipping protocol. In addition to **Validity** and **Agreement**, it guarantees **Probabilistic wait-free termination** with polynomial expected time for the asynchronous distributed model with communication via shared read/write variables in the presence of stopping failures.

The algorithm proceeds in rounds. Each process maintains two multiple-read single-write variables, recording its preferred *value* 1 or 2 (initially unknown, represented as 0), and its current *round*. The contents of the array *start* determines the initial preferences. Additional storage is needed to record copies of the preferred value and round of all other processes as observed by a given process; we use arrays *values* and *rounds* for this purpose. Note that the round number is unbounded, and due to asynchrony the processes may be in different rounds at any point in time. In Cadence SMV we have the following variable declarations:

```
#define N 10 /* number of processes (can be changed without affecting the proof) */
ordset PROC 1..N; /* set of process identifiers */
ordset NUM 0..; /* round numbers */
typedef PC {INITIAL, READ, CHECK, DECIDE, FAIL}; /* process phases */
act : PROC; /* the scheduler's choice of process */
start : array PROC of 1..2; /* start[i], initial preference of i */
pc : array PROC of PC; /* pc[i], the phase of process i */
value : array PROC of 0..2; /* value[i], current preference of i */
round : array PROC of NUM; /* round[i], current round number of i */
```

[2] See [11] for solutions based on read/modify/write variables, such as test-and-set.

```
values : array PROC of array PROC of 0..2;
/* values[i][j], j's preference when last read by i */
rounds : array PROC of array PROC of NUM;
/* rounds[i][j],  j's round number when last read by i */
count : array PROC of PROC; /* auxiliary counter for the reading loop */
```

The processes begin with the INITIALisation phase, where the unknown value is replaced with the preferred value from the array *start* and the round number is set to 1. Then each process repeatedly executes the READing then CHECKing phase until agreement. READing consists of reading the preferred value and round of all processes into the arrays *values* and *rounds*. Process i terminates in the CHECKing phase if it is a *leader* (i.e. its round is greater than or equal to that of any process) and if all processes whose round trails i's by at most 1 (i.e. are presumed not to have failed) agree. Otherwise, if all leaders agree, i updates its value to this preference, increments its round and returns to READing. In the remaining case, if i has a definite preference it "warns" that it may change by resetting it to 0 and returns to READing *without* changing its round number; if its preference is already 0, then i invokes a coin-flipping protocol to select a new value from $\{1, 2\}$ *at random*, increments its round number and returns to READing.

In Cadence SMV a simplified protocol (we have removed the possibility of FAILure for clarity) can be described as follows, where the random choice of preference from $\{1, 2\}$ has been replaced by a *nondeterministic assignment*:

```
switch (pc[act]) {
  INITIAL : {
    next(value[act]) := start[act];
    next(round[act]) := round[act] + 1;
    next(pc[act]) := READ; }
  READ : {
    next(pc[act]) := (count[act] = N) ? CHECK  :  READ;
    next(rounds[act][count[act]]) := round[count[act]];
    next(values[act][count[act]]) := value[count[act]];
    next(count[act]) := (count[act] = N) ? count[act]  :  count[act] + 1; }
  CHECK : {
    if (decide[act]) { /* all who disagree trail by two and I am a leader */
      next(pc[act]) := DECIDE;
    else if (agree[act][1] | agree[act][2]) { /* all leaders agree */
      next(pc[act]) := READ;
      next(count[act]) := 1;
      next(value[act]) := agree[act][1] ? 1  :  2; /* set value to leaders' preference */
      next(round[act]) := round[act] + 1; }
    else {
      next(pc[act]) := READ;
      next(count[act]) := 1;
      next(value[act]) := (value[act] > 0) ? 0  :  {1,2}; /* warn others or flip coin */
      next(round[act]) := (value[act] > 0) ? round[act]  :  round[act] + 1; } }
  }
```

where the missing formulas *decide* and *agree* are defined below, assuming that $j \in obs_i$ (process i has observed j) if either $j < count[i]$ or $pc[i] = CHECK$:

$agree[i][v]$ is true if, according to i, all leaders whose values have been read by process i agree on value v, where v is either 1 or 2; formally:

$$\begin{aligned} agree[i][v] &\overset{\text{def}}{=} \textstyle\bigwedge_j \, array\_agree[i][v][j] \\ array\_agree[i][v][j] &\overset{\text{def}}{=} j \in obs_i \rightarrow (rounds[i][j] \geq maxr[i] \rightarrow values[i][j] = v) \\ maxr[i] &\overset{\text{def}}{=} \max_{j \in obs_i} rounds[i][j] \end{aligned}$$

$decide[i]$ is true if, according to i, all that disagree trail by 2 or more rounds and i is a leader; formally:

$$\begin{aligned} decide[i] &\overset{\text{def}}{=} maxr[i] = round[i] \wedge (m1\_agree[i][1] \vee m1\_agree[i][2]) \\ m1\_agree[i][v] &\overset{\text{def}}{=} \textstyle\bigwedge_j \, array\_m1\_agree[i][v][j] \\ array\_m1\_agree[i][v][j] &\overset{\text{def}}{=} j \in obs_i \rightarrow (rounds[i][j] \geq maxr[i] - 1 \rightarrow values[i][j] = v) \end{aligned}$$

The above necessitates a variable, *maxr*, to store the maximum round number. The full protocol can be found at `www.cs.bham.ac.uk/~dxp/prism/consensus`.

It remains to provide a coin-flipping protocol which returns a preference 1 or 2, with a certain probability, whenever requested by a process in an execution. This could simply be a collection of N independent coins, one for each process, which deliver 1 or 2 with probability $\frac{1}{2}$ (independent of the current round). In [1] it is shown that such an approach yields *exponential* expected time to termination. The *polynomial expected time* is guaranteed by a *shared coin* protocol, which implements a collective random walk parameterised by the number of processes N and a constant $K > 1$ (independent of N). A new copy of this protocol is started *for each round*. The processes access a global shared counter, initially 0. On entering the protocol, a process flips a coin, and, depending on the outcome, increments or decrements the shared counter. Since we are working in a distributed scenario, several processes may simultaneously want to flip a coin, which is modelled as a *nondeterministic* choice between *probability distributions*, one for each coin flip. Note that several processes may be executing the protocol at the same time. Having flipped the coin, the process then reads the counter, say observing c. If $c \geq KN$ it chooses 1 as its preferred value, and if $c \leq -KN$ it chooses 2. Otherwise, the process flips the coin again, and continues doing so until it observes that the counter *has passed one of the barriers*. The barriers ensure that the scheduler cannot influence the outcome of the protocol by suspending processes that are about to move the counter in a given direction.

We denote by CF such a coin-flipping protocol and CF_r the collection of protocols, one for each round number r. Model checking of the shared coin protocol is described in Section 5.

3 The Proof Structure

Recall that to verify this protocol correct we need to establish the properties of **Validity**, **Agreement** and **Probabilistic wait-free termination**. The first two are independent of the actual values of probabilities. Therefore, we can verify these properties by *conventional model checking methods*, replacing the probabilistic choices with nondeterministic ones. In Section 4 we describe how we verify **Validity** and **Agreement** using the methods introduced in [12,13,14] for Cadence SMV.

We are left to consider **Probabilistic wait-free termination**. This property depends on the probability values with which either 1 or 2 is drawn, and, in particular, on the probabilistic properties of the coin-flipping protocol. However, there are several **probabilistic progress** properties which do *not* depend on any probabilistic assumptions. Similarly to the approach of [15] we analyse such properties in the non-probabilistic variant of the model, except we use Cadence SMV, thus *confining the probabilistic arguments* to a limited section of the analysis.

We now describe the outline of the proof based on [15]. First, we identify subsets of states of the protocol as follows: $\mathcal{D}$, the set of states in which all processes have decided; and $\mathcal{F}_v$, for $v \in \{1, 2\}$, the set of states where there exists $r \in \mathbb{N}$ and unique process i such that i's preferred *value* is v, i has just entered round r, and i is the only leader.

Non-probabilistic arguments: There are a number of non-probabilistic arguments, see [15]. We state the two needed to explain the main idea of the proof:

Invariant 1 From any state, the maximum round does not increase by more than 1 without reaching a state in $\mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{D}$.

Invariant 2 From any state of $\mathcal{F}_v$ with maximum round r, if in round r all processes leave the protocol CF_r agreeing on the value v, then the maximum round does not increase by more than 2 without reaching a state in $\mathcal{D}$.

These properties are independent of the probabilities of the coin-flipping protocol. So we can replace the random choices of CF with nondeterministic ones, except in round r where CF_r must return value v for all processes.

Probabilistic arguments: There are two probabilistic properties, listed below.

C1 For each fair execution of CF_r that starts with a reachable state of CF_r, *with probability 1* all processes that enter CF_r will eventually leave.

C2 For each fair execution of CF_r, and each value $v \in \{1, 2\}$, the probability that all processes that enter CF_r will eventually leave agreeing on the value v is at least p, where $0 < p \leq 1$.

Putting the arguments together: By **Invariant 1** and **C1** (since the coin-flipping protocol must return a value in order to continue), from *any* reachable state of the combined protocol, under any scheduling of nondeterminism, *with probability* 1 one can always reach a state either in $\mathcal{D}$, $\mathcal{F}_1$ or $\mathcal{F}_2$ such that the maximum round number increases by *at most* 1. Next by **Invariant 2**, **C1** and **C2**, from a state in $\mathcal{F}_v$, under any scheduling of nondeterminism, *with probability at least* p one can always reach a state in $\mathcal{D}$ with the maximum round number increasing by *at most 2*. Therefore, from these two properties, starting from *any* reachable state of the combined protocol, under any scheduling of nondeterminism, *with probability at least p* one can always reach a state in $\mathcal{D}$ (all processes have decided) with the maximum round number increasing by *at most* 3(=1+2).

It then follows that the expected number of rounds until $\mathcal{D}$ is reached is $O(\frac{1}{p})$. Thus, using independent coins where $p = \frac{1}{2^N}$ the expected number of rounds is $O(2^N)$. For the shared coin protocol, since $p = \frac{K-1}{2K}$, it is $O(1)$ (i.e. constant). This is because the round number does not increase while the processes perform the shared coin protocol. However, we must take account of the number of steps performed within the shared coin protocol; by random walk theory this yields expected time of $(K+1)^2N^2 = O(N^2)$ [1], which is indeed polynomial.

In the sequel we show how to use Cadence SMV and PRISM to mechanically verify the non-probabilistic and probabilistic arguments respectively. These have to be carried out at a low level, and therefore constitute the most tedious and error-prone part of the analysis. The remaining part of the proof, in which the separately verified arguments are put together, is not proved mechanically. It is sufficiently high level that it can be easily checked by hand. We believe that a fully mechanical analysis can be achieved with the help of a theorem prover.

4 The Cadence SMV Proof

Cadence SMV is a proof assistant which supports several *compositional methods* [12,13,14]. These methods permit the verification of large, complex, systems by reducing the verification problem to small problems that can be solved automatically by model checking. Cadence SMV provides a variety of such techniques including *induction*, *circular compositional reasoning*, *temporal case splitting* and *data type reduction*. For example, data type reduction is used to reduce large or *infinite* types to small finite types, and temporal case splitting breaks the proof into *cases* based on the value of a given variable. Combining data type reduction and temporal case splitting can reduce a complex proof to checking only a small number of simple subcases, thus achieving significant space savings.

There are two main challenges posed by the algorithm we consider: the round numbers are *unbounded*, leading to an *infinite* data type NUM, and we wish to prove the correctness for *any* number of processes, or, in other words, for *all* values of N. We achieve this by suitably combining data type reduction (`ordset`) with induction, circular compositional reasoning and temporal case splitting.

We briefly explain the `ordset` data type reduction implemented in Cadence SMV [14] with the help of the type NUM. For a given value r this reduction constructs an abstraction of this type shown in Figure 1, where the only constant is 0. The only operations permitted on this type are: equality/inequality testing (between abstract values), equality/inequality test against a constant, and increment/decrement the value by 1. For example, the following are allowed: comparisons $r > 0$ and $r = 0$ (but not $r = 1$) and $\texttt{next}(r) := r + 1$. With these restrictions on the operations, the abstract representations as shown in Figure 1 are *isomorphic* for all $r \in NUM$. Therefore, it suffices to check a property for a *single* value of r. The reduction of the data type *PROC* is similar, except that there are two constants, 1 and N; see [14] for more detail.

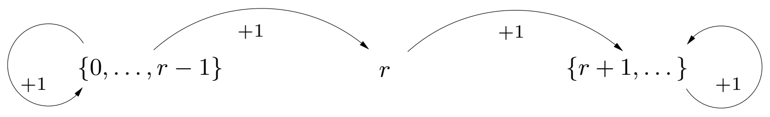

Fig. 1. Abstraction of *NUM*

We now illustrate the `ordset` reduction with a simple property, concerning the *global maximum round*, that is, the maximum round number over all processes. In Cadence SMV we can define this as follows:

`next`$(gmaxr)$:= `next`$(round[act]) > gmaxr$? `next`$(round[act])$: $gmaxr$;

However, since *act* ranges over *PROC*, the value of *gmaxr* depends on *all* instances $round[i]$ for $i \in N$. We therefore introduce a *history* variable which records the value of $round[act]$ and replaces the implicit dependence on N with a dependence on a single variable. We redefine *gmaxr* as follows:

`next`$(hist)$:= `next`$(round[act])$;
`next`$(gmaxr)$:= `next`$(hist) > gmaxr$? `next`$(hist)$: $gmaxr$;

We can now state that *gmaxr* is indeed the global maximum round number:

`forall` $(i$ `in` $PROC)$ $max[i]$: `assert G` $(round[i] \leq gmaxr)$;

To prove this holds, we *case split* on the value of $round[i]$ and suppose that $max[i]$ holds at time $t-1$. Furthermore, by setting the variables that do not affect the satisfaction of $max[i]$ to be *free* (allowing these variables to range over *all* the possible values of their types), we can improve the efficiency of model checking by a factor of 10. Though perhaps not important for this simple property, such improvements are crucial for more complex properties, as without freeing certain variables model checking quickly becomes infeasible. The proof is:

```
forall (r in NUM) {
    subcase max[i][r] of max[i] for round[i] = r; /* case split on round[i] */
    using (max[i]), /* assume max[i] holds at time t − 1 */
    agree//free, decide//free, start//free, value//free, /* free variables */
    prove max[i] };
```

Through the `ordset` data type reduction SMV reduces this proof to checking $max[i][r]$ for a single value of i (=2) and a single value of r (=1).

The full proof of **Validity**, **Agreement** and **Non-probabilistic progress** is available at `www.cs.bham.ac.uk/~dxp/prism/consensus`. The proof consists of approximately 50 lemmas, requiring at most 270 MB of memory[3]. Judicious choice of data reduction/freeing is important, as otherwise SMV may return false, but SMV allows one to inspect the cone of influence to identify the variables that are used in the proofs.

[3] The version of Cadence SMV we have used is not fully compatible with the release of 08.08.00.

4.1 Proof of Validity

We now outline the proof of **Validity**, which we verify by proving the contrapositive: if no process starts with value v then no process decides on v. For simplicity suppose $v = 2$. The hypothesis is that no process starts with value 2:

```
forall ($i$ in $PROC$) $valid\_assump[i]$ : assert G $\neg($ $start[i] = 2$ $)$;
```

which is assumed throughout the proof, and the conclusion is:

```
forall ($i$ in $PROC$) $validity[i]$ : assert G$(pc[i] = DECIDE \rightarrow \neg(value[i] = 2))$;
```

The important step in proving *validity* is seeing that if all processes start with preference 1, then any process i past its INITIAL phase, i.e. whose round number is positive, has preferred value 1 and the predicate $agree[i][1]$ holds. To prove *validity* we therefore first prove the stronger properties:

```
forall ($i$ in $PROC$) {
    $valid1[i]$ : assert G $(round[i] > 0 \rightarrow value[i] = 1)$;
    $valid2[i]$ : assert G $(round[i] > 0 \rightarrow agree[i][1])$; }
```

We prove $valid1[i]$ by case splitting on $round[i]$ and assuming $valid2[i]$ holds at time $t-1$. Also, since $round[i] = 0$ is a special case, we must add 0 to the abstraction of NUM (otherwise Cadence SMV returns false), i.e. NUM is abstracted to $0, \{1, \ldots, r-1\}, r, \{r+1, \ldots\}$. The proof in Cadence SMV has the following form:

```
forall ($r$ in $NUM$) {
    subcase $valid1[i][r]$ of $valid1[i]$ for $round[i] = r$;
    using $valid\_assump[i]$, $(valid2[i])$, $NUM \rightarrow \{0, r\}, \ldots,$ prove $valid1[i][r]$; }
```

To prove $valid2[i]$, we have the additional complication of $agree[i][1]$ being defined as the conjunction of an array ($array\_agree[i][1][j]$ for $j \in PROC$), which again contains an implicit dependency on all values of the set $PROC$. Instead, we consider each element of the array separately. In particular, we first prove the auxiliary property $valid3[i]$ elementwise, assuming *valid1* holds, and again add 0 to the abstraction of NUM:

```
forall ($i$ in $PROC$) forall ($j$ in $PROC$) {
    $valid3[i][j]$ : assert G ( $round[i] > 0 \rightarrow array\_agree[i][1][j]$ );
    forall ($r$ in $NUM$) {
        subcase $valid3[i][j][r]$ of $valid3[i][j]$ for $maxr[i] = r$;
        using $valid\_assump[j]$, $valid1[j]$, $NUM \rightarrow \{0, r\}, \ldots,$ prove $valid3[i][j][r]$; }}
```

Next we use $valid3[i][j]$ to prove $valid2[i]$ through a proof by *contradiction*: first consider the processes j such that $array\_agree[i][1][j]$ is false:

```
forall ($i$ in $PROC$) $y[i] := \{ j : j$ in $PROC, \neg array\_agree[i][1][j] \}$;
```

Then we consider a particular $j \in y[i]$ when proving $valid2[i]$ while using the fact that $valid3[i][j]$ holds:

```
forall ($j$ in $PROC$) {
    subcase $valid2[i][j]$ of $valid2[i]$ for $y[i] = j$;
    using $valid3[i][j], \ldots,$ prove $valid2[i][j]$; }
```

The contradiction then arises since, by $valid3[i][j]$, $array\_agree[i][1][y[i]]$ must be true. The apparent circularity between these properties is broken since *valid1* assumes *valid2* at time $t-1$.

4.2 Proof of Agreement

We now outline the proof of Invariant 6.3 of [15] which is used to prove **Agreement**, the most difficult of the requirements. First we define new predicates $\mathit{fill\_maxr}[i]$, $\mathit{array\_fill\_agree}[i][v][j]$ and $\mathit{fill\_agree}[i][v]$ to be the same as the corresponding predicates $\mathit{maxr}[i]$, $\mathit{array\_agree}[i][v][j]$ $\mathit{agree}[i][v]$, except an incomplete observation of a process is "filled in" with the actual values of the unobserved processes. More formally:

$$\mathit{fill\_rounds}[i][j] \overset{\text{def}}{=} \texttt{if } j \in \mathit{obs}_i \texttt{ then } \mathit{rounds}[i][j] \texttt{ else } \mathit{round}[j]$$
$$\mathit{fill\_values}[i][j] \overset{\text{def}}{=} \texttt{if } j \in \mathit{obs}_i \texttt{ then } \mathit{values}[i][j] \texttt{ else } \mathit{value}[j].$$

Invariant 6.3 of [15]. *Let i be a process. Given a reachable state, let $v = \mathit{value}[i]$. If $\mathit{fill\_maxr}[i] = \mathit{round}[i]$, $\mathit{m1\_agree}[i][v]$ and $\mathit{fill\_agree}[i][v]$, then*

a. $\forall_j\ \mathit{agree}[j][v]$
b. $\forall_j\ \mathit{round}[j] \geq \mathit{round}[i] \rightarrow \mathit{value}[j] = v$
c. $\forall_{j \in \mathit{obs}_i}\ (\mathit{round}[j] = \mathit{round}[i] - 1 \wedge \mathit{value}[j] \neq v) \rightarrow \mathit{fill\_maxr}[j] \leq \mathit{round}[i]$.

We now describe our approach to proving Invariant 6.3. For simplicity, we have restricted our attention to when $v = 1$. To ease the notation we let:

$$C[i] \overset{\text{def}}{=} (\mathit{fill\_maxr}[i] = \mathit{round}[i]) \wedge \mathit{m1\_agree}[i][1] \wedge \mathit{fill\_agree}[i][1] \wedge (\mathit{value}[i] = 1).$$

We first split Invariant 6.3 into separate parts corresponding to the conditions a, b and c. The main reason for this is that the validity of the different cases depends on different variables of the protocol. We are therefore able to "free" more variables when proving the cases separately, and hence improve the efficiency of the model checking. Formally, conditions a and b of Invariant 6.3 are given by:

```
forall (i in PROC) forall (j in PROC)
    inv63a[i][j] : assert G ( C[i] → agree[j][1] );
    inv63b[i][j] : assert G ( C[i] → (round[j] ≥ round[i] → value[j] = 1) );
```

Note that, when proving $\mathit{inv63a}[i][j]$, $\mathit{agree}[j][1]$ is the conjunction of an array. We therefore use the same proof technique as outlined for $\mathit{valid2}[i]$ in Section 4.1, that is, we first prove:

```
forall (i in PROC) forall (j in PROC) forall (k in PROC)
    inv63ak[i][j][k] : assert G ( C[i] → array_agree[j][1][k] );
```

We encounter a similar problem with the precondition, $C[i]$, since $\mathit{m1\_agree}[i][1]$ and $\mathit{fill\_agree}[i][1]$ are conjunctions of arrays. In this case, we use a version of Lemma 6.12 of [15]. Informally, this lemma states: if $C[i]$ holds in the *next state* and the transition to reach this state does *not* involve process i changing the value of $\mathit{round}[i]$ or $\mathit{value}[i]$, then $C[i]$ holds in the *current state*. More precisely, we have the following properties:

```
forall (i in PROC) {
    lem612a[i] : assert G ( (¬(act = i) ∧ X (C[i])) → (C[i]) );
    lem612b[i] : assert G ( (act = i ∧ (pc[i] = READ) ∧ X (C[i])) → (C[i]) );
    lem612c[i] : assert G ( (act = i ∧ X ((pc[i] = DECIDE) ∧ C[i])) → (C[i]) ); }
```

When proving $inv63ak[i][j][k]$ we case split on $round[i]$ and assume $inv63ak[i][j][k]$ and $inv63b[i][k]$ hold at time $t-1$ (Invariant 6.3c is not needed). Additional assumptions include those of Lemma 6.12 given above. Also, since $m1\_agree[i]$ involves $r-1$ where r is of type NUM, we abstract NUM to $\{0,\ldots,r-2\}, r-1$, $r, \{r+1,\ldots\}$. The actual proof in Cadence SMV has the following form:

```
forall (r in NUM) {
    subcase inv63ak[i][j][k][r] of inv63ak[i][j][k] for round[i] = r;
    using (inv63ak[i][j][k]), (inv63b[i][k]), lem612a[i], lem612b[i], lem612c[i],
    NUM → {r − 1 . . r}, . . . , prove inv63ak[i][j][k][r]; }
```

5 Verification with PRISM

PRISM, a Probabilistic Symbolic Model Checker, is an experimental tool described in [6], see `www.cs.bham.ac.uk/~dxp/prism`. It is built in Java/C++ using the CUDD [16] package which supports MTBDDs. The system description language of the tool is a probabilistic variant of Reactive Modules. The specifications are given as formulas of the probabilistic temporal logic PCTL [8,4]. PRISM builds a symbolic representation of the model as an MTBDD and performs the analysis implementing the algorithms of [3,4]. It supports a symbolic engine based on MTBDDs as well as a sparse matrix engine.

A summary of experimental results obtained from the shared coin-flipping protocol modelled and analysed using the MTBDD engine is included in the table below. Further details, including the description of the coin-flipping protocol, can be found at `www.cs.bham.ac.uk/~dxp/prism/consensus`. Both properties **C1** and **C2** are expressible in PCTL. **C1** is a probability 1 property, and therefore admits efficient *qualitative* [17] probabilistic analysis such as the probability-1 precomputation step [6], whereas **C2**, on the other hand, is *quantitative*, and requires calculating the minimum probability that, starting from the *initial state* of the coin-flipping protocol, *all* processes leave the protocol agreeing on a given value. Our analysis is mechanical, and demonstrates that the analytical lower bound $\frac{K-1}{2K}$ obtained in [1] is reasonably tight (the discrepancy is greater for smaller values of K, not included).

N	K	#states	construction time (s):	**C1**	**C2**		
				time (s):	time (s):	probability:	bound $(K-1)/2K$:
2	64	8,208	1.108	0.666	3689	0.493846	0.4921875
4	32	329,856	2.796	6.497	212784	0.494916	0.484375
8	16	437,194,752	54.881	59.668	1085300	0.47927	0.46875
10	8	10,017,067,008	26.719	139.535	986424	0.4463	0.4375

Fig. 2. Model checking of the coin-flipping protocol

6 Conclusion

In this paper we have for the first time mechanically verified a complex randomized distributed algorithm, thus replacing tedious proofs by hand of a large numbers of lemmas with manageable, re-usable, and efficient proofs with Cadence SMV and an automatic check of the probabilistic properties with PRISM. The verification of the protocol is fully mechanised at the low level, while some simple high-level arguments are carried out manually. A fully automated proof can be achieved by involving a theorem prover for the manual part of the analysis. We believe that the techniques introduced here are applicable more generally, for example, to analyse [10,5].

Acknowledgements: We are grateful to Ken McMillan for supplying a recent version of Cadence SMV and suggesting appropriate proof methods.

References

1. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, 1990.
2. C. Baier, E. Clarke, and V. Hartonas-Garmhausen. On the semantic foundations of Probabilistic VERUS. In C. Baier, M. Huth, M. Kwiatkowska, and M. Ryan, editors, *Proc. PROBMIV'98*, volume 22 of *ENTCS*, 1998.
3. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11:125–155, 1998.
4. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Proc. FST & TCS*, volume 1026 of *LNCS*, pages 499–513, 1995.
5. C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. In *Proc. PODC'00*, pages 123–132, 2000.
6. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic systems using MTBDDs and the Kronecker representation. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS'2000*, volume 1785 of *LNCS*, pages 395–410, 2000.
7. M. Fischer, N. Lynch, and M.Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(5):374–382, 1985.
8. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(4):512–535, 1994.
9. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov Chain Model Checker. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS 2000*, volume 1785 of *LNCS*, pages 347–362, 2000.
10. A. Itai and M. Rodeh. The lord of the ring or probabilistic methods for breaking symmetry in distributed networks. Technical Report RJ 3110, IBM, 1981.
11. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
12. K. McMillan. Verfication of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *Proc. CAV'98*, volume 1427 of *LNCS*, pages 110–121, 1998.

13. K. McMillan. Verification of infinite state systems by compositional model checking. In L. Pierre and T. Kropf, editors, *Proc. CHARME'99*, volume 1703 of *LNCS*, pages 219–233, 1999.
14. K. McMillan, S. Qadeer, and J. Saxe. Induction and compositional model checking. In E. Emerson and A. P. Sistla, editors, *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 312–327, 2000.
15. A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing*, 13(3):155–186, 2000.
16. F. Somenzi. CUDD: CU decision diagram package. Public software, Colorado University, Boulder, 1997.
17. M. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proc. FOCS'85*, pages 327–338, 1985.