A Choreographic Language for PRISM

... Author: Please enter affiliation as second parameter of the author macro

... Author: Please enter affiliation as second parameter of the author macro

Abstract

This is the abstract

2012 ACM Subject Classification Theory of computation → Type theory; Computing methodologies → Distributed programming languages; Theory of computation → Program verification

Keywords and phrases Session types, PRISM, Model Checking

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.m

Funding This work was supported by

1 Formal Languages

This section provides the formal definition of our choreographic language as well as process algebra representing PRISM [**?**].

1.1 PRISM

We start by describing PRISM semantics. To the best of our knowledge, the only formalisation of a semantics for PRISM can be found on the PRISM website [**?**]. Our approach starts from this and attempts to make more precise some informal assumptions and definitions.

Syntax. Let p range over a (possibly infinite) set of module names $\mathcal{R}$, a over a (possibly infinite) set of labels $\mathcal{L}$, x over a (possibly infinite) set of variables Var, and v over a (possibly infinite) set of values Val. Then, the syntax of the PRISM language is given by the following grammar:

$$
\begin{array}{llcll}
\text{(Networks)} & N, M & ::= & \mathbf{0} & \text{empty network} \\
 & & \mid & \mathsf{p} : \{F_i\}_i & \text{module} \\
 & & \mid & M|[A]|M & \text{parallel composition} \\
 & & \mid & M/A & \text{action hiding} \\
 & & \mid & \sigma M & \text{substitution} \\
\\
\text{(Commands)} & F & ::= & [a]g \to \Sigma_{i \in I}\{\lambda_i : u_i\} & g \text{ is a boolean expression in } E \\
\\
\text{(Assignment)} & u & ::= & (x' = E) & \text{update } x\text{, element of } \mathcal{V}\text{, with } E \\
 & & \mid & A\&A & \text{multiple assignments} \\
\text{(Expr)} & E & ::= & f(\tilde{E}) \quad \mid \quad x \quad \mid \quad v &
\end{array}
$$

Networks are the top syntactic category for system of modules composed together. The term $\mathbf{0}$ represent an empty network. A module is meant to represent a process running in the system, and is denoted by its variables and its commands. Formally, a module $\mathsf{p} : \{F_i\}_i$ is identified by its name p and a set of commands F_i. Networks can be composed in parallel, in a CSP style: a term like $M_1|[A]|M_2$ says that networks M_1 and M_2 can interact with each other using labels in the finite set A. The term M/A is the standard CSP/CCS hiding operator. Finally σM is equivalent to applying the substitution σ to all variables in x. A substitution is a function that given a variable returns a value. When we write σN we refer to the term obtained by replacing every free variable x in N with $\sigma(x)$. Marco: Is this really the way substitution is used? Where does it become important? Commands in a module have the form $[a]g \to \Sigma_{i\in I}\{\lambda_i : u_i\}$. The label a is used for synchronisation (it is a condition that allows the command to be executed when all other modules having a command on the same label also execute). The term g is a guard on the current variable state. If both label and the guards are enabled, then the command executes in a probabilistic way one of the branches. Depending on the model we are going to use, the value λ_j is either a real number representing a rate (when adapting an exponential distribution) or a probability. If we are using probabilities, then we assume that terms in every choice are such that the sum of the probabilities is equal to 1.

Semantics. In order to give a probabilistic semantics to PRISM, we proceed by steps. First, we define $\{\![-]\!\}$, as the closure of the following rules:

$$\frac{}{F_i \in \{\![\mathsf{p} : \{F_i\}_i]\!\}}\ (\mathsf{Module}) \qquad \frac{[]E \to \{\lambda_i : x_i = E_i\}_{i\in I} \in \{\![M_j]\!\} \quad j \in \{1,2\}}{[]E \to \{\lambda_i : x_i = E_i\}_{i\in I} \in \{\![M_1|[A]|M_2]\!\}}\ (\mathsf{Par}_1)$$

$$\frac{[a]E \to \{\lambda_i : x_i = E_i\}_{i\in I} \in \{\![M_j]\!\} \quad a \notin A \quad j \in \{1,2\}}{[a]E \to \{\lambda_i : x_i = E_i\}_{i\in I} \in \{\![M_1|[A]|M_2]\!\}}\ (\mathsf{Par}_2)$$

$$\frac{[a]E \to \{\lambda_i : x_i = E_i\}_{i\in I} \in \{\![M_1]\!\} \quad [a]E' \to \{\lambda'_j : y_j = E'_j\}_{j\in J} \in \{\![M_2]\!\} \quad a \in A}{[]E \wedge E' \to \{\lambda_i * \lambda'_j : x_i = E_i \wedge y_j = E'_j\}_{i\in I, j\in J} \in \{\![M_1|[A]|M_2]\!\}}\ (\mathsf{Par}_3)$$

$$\frac{[]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\}}{[]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M/A]\!\}}\ (\mathsf{Hide}_1) \qquad \frac{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\} \quad a \notin A}{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M/A]\!\}}\ (\mathsf{Hide}_2)$$

$$\frac{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\} \quad a \in A}{[]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M/A]\!\}}\ (\mathsf{Hide}_3) \qquad \frac{[]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\}}{[]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![\sigma M]\!\}}\ (\mathsf{Subst}_1)$$

$$\frac{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\} \quad a \notin \mathsf{dom}(\sigma)}{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![\sigma M]\!\}}\ (\mathsf{Subst}_2)$$

$$\frac{[a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![M]\!\} \quad a \in \mathsf{dom}(\sigma)}{[\sigma a]E \to \{\lambda_j : x_i = E_i\}_{i\in I} \in \{\![\sigma M]\!\}}\ (\mathsf{Subst}_3)$$

The rules above work with modules, parallel composition, name hiding, and substitution. The idea is that given a network, we wish to collect all those commands F that are contained in the network, independently from which module they are being executed in. Intuitively, we can regard $\{\![N]\!\}$ as a set, where starting from all commands present in the syntax, we do some filtering and renaming, based on the structure of the network.

Now, given $\{\![N]\!\}$, we define a transition system that shows how the system evolves. Let state be a function that given a variable in Var returns a value in Val. Then, given an initial state state_0, we can define a transition system where each of node is a (different) state function. Then, we can move from state_1 to state_2 whenever ... Formally, a transition system is defined as:

▶ **Definition 1** (Transition System). *[put definition of transition system here.]*

We can then define a transition system $\mathcal{T} = (2^{\mathsf{state}}, \mathsf{state}_0, \ldots)$ [fix details here].

1.2 Choreographies

Syntax. Our choreographic language is defined by the following syntax:

$$\text{(Chor)} \quad C \;::=\; \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma\{\lambda_j : x_j = E_j;\; C_j\}_{j \in J} \;\mid\; EpC_1C_2 \;\mid\; X \;\mid\; \mathbf{0}$$

We comment the various constructs. The syntactic category C denotes choreographic programmes. The term $\mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma\{\lambda_j : x_j = E_j;\; C_j\}_{j \in J}$ denotes an interaction initiated by role p with roles p_i. Unlike in PRISM, a choreography specifies what interaction must be executed next, shifting the focus from what can happen to what must happen. When the synchronisation happens then, in a probabilistic way, one of the branches is selected as a continuation. The term $\mathsf{if}\ E@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ factors in some local choices for some particular roles. [write a bit more about procedure calls, recursion and the zero process]

1.3 Projection from Choreographies to PRISM

Mapping Choreographies to PRISM. We need to run some standard static checks because, since there is branching, some terms may not be projectable.

$$\begin{array}{l}
\big(q \in \{\mathsf{p}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}\big) \\
\mathsf{proj}(q, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma\{\lambda_j : x_j = E_j;\; C_j\}_{j \in J}, s) = \\
\qquad \{[l] s_{\mathsf{p}_1} = s \to \Sigma_{i \in I}\{\lambda_i ::_i\} s_{\mathsf{p}_1} = s_{\mathsf{p}_1} + 1, [l] s_{\mathsf{p}_1} = s \to \Sigma_{i \in I}\{\lambda_i ::_i\} s_{\mathsf{p}_1} = s_{\mathsf{p}_1} + 1\} \quad \cup \\
\qquad \mathsf{proj}(\mathsf{p}_1, C_1, s+1) \quad \cup \quad \mathsf{proj}(\mathsf{p}_1, C_2, s + \mathsf{depth}(C_1))
\end{array}$$

$$\begin{array}{l}
\big(q \notin \{\mathsf{p}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}\big) \\
\mathsf{proj}(q, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma\{\lambda_j : x_j = E_j;\; C_j\}_{j \in J}, s) \;=\; \mathsf{proj}(\mathsf{p}_1, C_1, s) \;\cup\; \mathsf{proj}(\mathsf{p}_1, C_2, s + \mathsf{depth}(C_1))
\end{array}$$

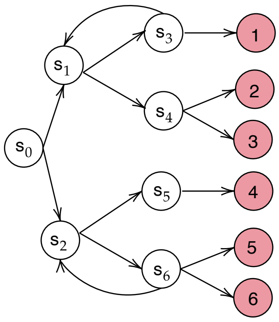
$$\begin{array}{l}
(q = \mathsf{p}) \\
\mathsf{proj}(q, EpC_1C_2, s) = \\
\qquad \{[] s_{\mathsf{p}_1} = s \& E \to \Sigma_{i \in I}\{\lambda_i ::_i\} s_{\mathsf{p}_1} = s_{\mathsf{p}_1} + 1, [] s_{\mathsf{p}_1} = s \& \mathsf{not}(E) \to \Sigma_{i \in I}\{\lambda_i ::_i\} s_{\mathsf{p}_1} = s_{\mathsf{p}_1} + 1\} \quad \cup \\
\qquad \mathsf{proj}(\mathsf{p}_1, C_1, s+1) \quad \cup \quad \mathsf{proj}(\mathsf{p}_1, C_2, s + \mathsf{depth}(C_1))
\end{array}$$

2 Tests

In this section we present our experimental evaluation of our language. We focus on four benchmarks: the dice program and the random graphs protocol that we compare with the test cases reported in the PRISM repository[1]; the Bitcoin proof of work protocol and the Hybrid Casper protocol, presented in [2, 4].

2.1 The Dice Program

The first test case we focus on the Dice Program[2][5]. The following program models a die using only fair coins. Starting at the root vertex (state s_0), one repeatedly tosses a coin. Every time heads appears, one takes the upper branch and when tails appears, the lower branch. This continues until the value of the die is decided.



In Listing 1, we report the modelled program using the choreographic language while in Listing 2 the generated PRISM program is shown.

```
preamble
"dtmc"
endpreamble

n = 1;

Dice → Dice : "d : [0..6] init 0;" ;

{
DiceProtocol0 := Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol1
                                +["0.5*1"] " "&&" " . DiceProtocol2)

DiceProtocol1 := Dice → Dice : (+["0.5*1"] " "&&" " .
                        Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol1
                                       +["0.5*1"] "(d'=1)"&&" " . DiceProtocol3)
                                  +["0.5*1"] " "&&" " .
                        Dice → Dice : (+["0.5*1"] "(d'=2)"&&" " . DiceProtocol3
                                        +["0.5*1"] "(d'=3)"&&" " . DiceProtocol3))

DiceProtocol2 := Dice → Dice : (+["0.5*1"] " "&&" " .
                        Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol2
                                        +["0.5*1"] "(d'=4)"&&" " . DiceProtocol3)
                               +["0.5*1"] " "&&" " .
                        Dice → Dice : (+["0.5*1"] "(d'=5)"&&" " . DiceProtocol3
                                       +["0.5*1"] "(d'=6)"&&" " . DiceProtocol3))

DiceProtocol3 := Dice → Dice : (["1*1"] " "&&" ".DiceProtocol3)
}
```

[1] https://www.prismmodelchecker.org/casestudies/
[2] https://www.prismmodelchecker.org/casestudies/dice.php

Listing 1 Choreographic language for the Dice Program.

```
dtmc

module Dice
       Dice : [0..11] init 0;
       d : [0..6] init 0;

       [] (Dice=0) → 0.5 : (Dice'=2) + 0.5 : (Dice'=6);
       [] (Dice=2) → 0.5 : (Dice'=3) + 0.5 : (Dice'=4);
       [] (Dice=3) → 0.5 : (Dice'=2) + 0.5 : (d'=1)&(Dice'=10);
       [] (Dice=4) → 0.5 : (d'=2)&(Dice'=10) + 0.5 : (d'=3)&(Dice'=10);
       [] (Dice=6) → 0.5 : (Dice'=7) + 0.5 : (Dice'=8);
       [] (Dice=7) → 0.5 : (Dice'=6) + 0.5 : (d'=4)&(Dice'=10);
       [] (Dice=8) → 0.5 : (d'=5)&(Dice'=10) + 0.5 : (d'=6)&(Dice'=10);
       [] (Dice=10) → 1 : (Dice'=10);

endmodule
```

Listing 2 Generated PRISM program for the Dice Program.

By comparing our model with the one presented in the PRISM documentation, we notice that the difference is the number assumed by the variable `Dice`. In particular, the variable assumes different values and this is due to how the generation in presence of a branch is done. However, this does not cause any problems since the updates are done correctly and the states are unique. Moreover, to prove the generated program is correct, we show that the probability of reaching a state where

$$\texttt{d=k} \text{ for } \texttt{k} = 1, \ldots, 6 \text{ is } 1/6.$$

The results are displayed in Figure 1, where we compare the probability we obtain with our generated model and the one obtained with the original PRISM model. As expected, the results are equiivalent.

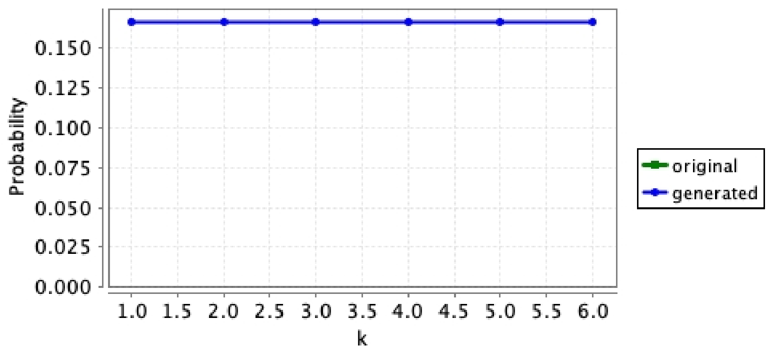
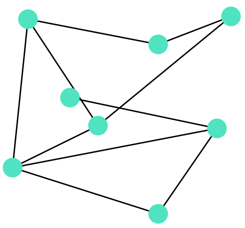


Figure 1 Probability of reaching a state where $d = k$, for $k = 1, \ldots, 6$.

2.2 Random Graphs Protocol



The second case study we report is the random graphs protocol presented in the PRISM documentation[3]. It investigates the likelihood that a pair of nodes are connected in a random graph. More precisely, we take into account the the set of random graphs $G(n, p)$, i.e. the set of random graphs with n nodes where the probability of there being an edge between any two nodes equals p.

The model is divided in two parts: at the beginning the random graph is built. Then the algorithm finds nodes that have a path to node 2 by searching for nodes for which one can reach (in one step) a node for which the existence of a path to node 2 has already been found.

The choreographic model is shown in Listing 3, while in Listing 4, we report only part of the generated PRISM module (the modules M_2, M_3 and P_2, P_3 are equivalent to, respectively, M_1 and P_2 and can be found in the repository[4]).

```
preamble
"mdp"
"const double p;"
endpreamble

n = 3;

PC -> PC : " ";
M[i] -> i in [1...n] M[i] : "varM[i] : bool;";
P[i] -> i in [1...n] P[i] : "varP[i] : bool;";

{
GraphConnected0 :=
        PC -> M[i] : (+["1*p"] " "&&"(varM[i]'=true)". END
                      +["1*(1-p)"] " "&&"(varM[i]'=false)". END)
        PC -> P[i] : (+["1*p"] " "&&"(varP[i]'=true)" . END
                      +["1*(1-p)"] " "&&"(varP[i]'=false)".
                      if "(PC=6)&!varP[i]&((varP[i] & varM[i]) | (varM[i+1] & varP[
                          ↪ i+2])) "@P[i] then {
                              ["1"]"(varP[i]'=true)"@P[i] . GraphConnected0
                      })
}
```

Listing 3 Choreographic language for the Random Graphs Protocol.

```
mdp
const double p;

module PC
   PC : [0..7] init 0;

```

[3] https://www.prismmodelchecker.org/casestudies/graph_connected.php
[4] https://github.com/adeleveschetti/choreography-to-PRISM

```
    [DPPGR] (PC=0) → 1 : (PC'=1);
    [YCJJG] (PC=1) → 1 : (PC'=2);
    [TWGVA] (PC=2) → 1 : (PC'=3);
    [NODPZ] (PC=3) → 1 : (PC'=4);
    [FDALJ] (PC=4) → 1 : (PC'=5);
    [DCKXC] (PC=5) → 1 : (PC'=6);
endmodule

module M1
    M1 : [0..1] init 0;
    varM1 : bool;

    [DPPGR] (M1=0) → p :(varM1'=true)&(M1'=0) + (1-p) :(varM1'=false)&(M1'=0);
endmodule

...

module P1
    P1 : [0..3] init 0;
    varP1 : bool;

    [NODPZ] (P1=0) → p:(varP1'=true)&(P1'=0) + (1-p):(varP1'=false)&(P1'=0);
    [] (P1=0)&(PC=6)&!varP1&((varP1 & varM1) | (varM2& varP3))
                                → 1 : (varP1'=true)&(P1'=0);
endmodule
...
```

Listing 4 Generated PRISM program for the Random Graphs Protocol.

The model is very similar to the one presented in the PRISM repository, the main difference is that we use state variables also for the modules $\mathtt{P}_i$ and $\mathtt{M}_i$, where in the original model they were not requires. However, this does not affect the behaviour of the model, as the reader can notice from the results of the probability that nodes 1 and 2 are connected showed in Figure 2.

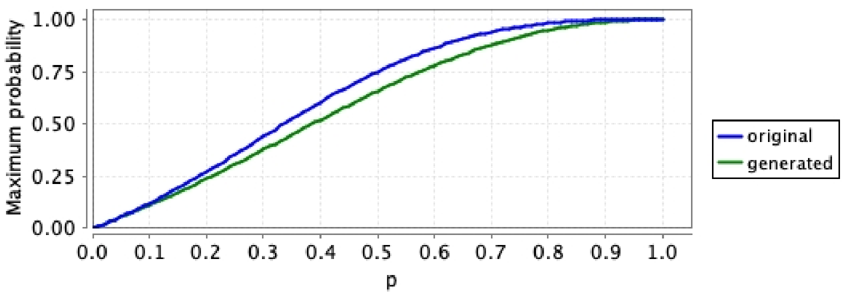
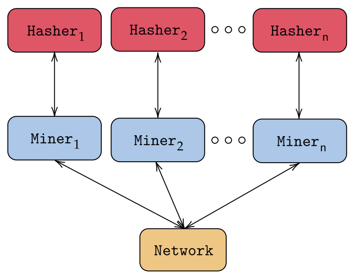


Figure 2 Probability that the nodes 1 and 2 are connected.

2.3 Proof of Work Bitcoin Protocol

In [2], the authors decided to extend the PRISM model checker with dynamic data types in order to model the Proof of Work protocol implemented in the Bitcoin blockchain [6].



The Bitcoin system is the result of the parallel composition of n Miner processes, n *Hasher* processes and a process called *Network*. In particular:

- The *Miner* processes model the blockchain mainers that create new blocks and add them to their local ledger;
- the *Hasher* processes model the attempts of the miners to solve the cryptopuzzle;
- the *Network* process model the broadcast communication among miners.

Since we are not interested in the properties obtained by analyzing the protocol, we decided to consider $n = 4$ miner and hasher processes; the model can be found in Listing 5.

```
preamble
...
endpreamble

n = 4;

...

{
PoW := Hasher[i] -> Miner[i] :
(+["mR*hR[i]"] " "&&"(b[i]'=createB(b[i],B[i],c[i]))&(c[i]'=c[i]+1)" .
       Miner[i] -> Network :
              (["rB*1"] "(B[i]'=addBlock(B[i],b[i]))" &&
              foreach(k != i) "(set[k]'=addBlockSet(set[k],b[i]))" @Network .PoW)
 +["lR*hR[i]"] " " && " " .
       if "!isEmpty(set[i])"@Miner[i] then {
              ["r"] "(b[i]'=extractBlock(set[i]))"@Miner[i] .
                     Miner[i] -> Network :
                     (["1*1"] "(setMiner[i]' = addBlockSet(setMiner[i] , b[i]))"&&
                         "(set[i]' = removeBlock(set[i],b[i]))" . PoW)
       }
       else{
              if "canBeInserted(B[i],b[i])"@Miner[i] then {
                     ["1"] "(B[i]'=addBlock(B[i],b[i]))&&(setMiner[i]'=removeBlock
                         (setMiner[i],b[i]))"@Miner[i] . Pow
              }
              else{
                     PoW
              }
       }
)
}
```

Listing 5 Choreographic language for the Proof of Work Bitcoin Protocol.

Part of the generated PRISM code is shown in Listing 6, the modules $Miner_2$, $Miner_3$, $Miner_4$ and $Hasher_2$, $Hasher_3$, $Hasher_4$ are equivalent to $Miner_1$ and $Hasher_1$, respectively. Our generated PRISM model is more verbose than the one presented in [2], this is due to the fact that for the `if-then-else` expression, we always generate the `else` branch. and this leads to having more instructions

```
...

module Miner1
   Miner1 : [0..7] init 0;
   b1 : block {m1,0;genesis,0} ;
   B1 : blockchain [{genesis,0;genesis,0}];
   c1 : [0..N] init 0;
   setMiner1 : list [];

   [PZKYT] (Miner1=0) → hR1 : (b1'=createB(b1,B1,c1))&(c1'=c1+1)&(Miner1'=1);
   [EUBVP] (Miner1=0) → hR1 : (Miner1'=2);
   [HXYKO] (Miner1=1) → 1 : (B1'=addBlock(B1,b1))&(Miner1'=0);
   [] (Miner1=2)&!isEmpty(set1) → r : (b1'=extractBlock(set1))&(Miner1'=4);
   [SRKSV] (Miner1=4) → 1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Miner1'=0)
       ↪ ;
   [] (Miner1=2)&!(!isEmpty(set1)) → 1 : (Miner1'=5);
   [] (Miner1=5)&canBeInserted(B1,b1) → 1 : (B1'=addBlock(B1,b1))&(setMiner1'=
       ↪ removeBlock(setMiner1,b1))&(Miner1'=0);
   [] (Miner1=5)&!(canBeInserted(B1,b1)) → 1 : (Miner1'=0);

endmodule
...
module Network
Network : [0..1] init 0;
   set1 : list [];
   ...

   [HXYKO] (Network=0) → 1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,
       ↪ b3))&(set4'=addBlockSet(set4,b4))&(Network'=0);
   [SRKSV] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
   ...

endmodule

module Hasher1
Hasher1 : [0..1] init 0;

[PZKYT] (Hasher1=0) → mR : (Hasher1'=0);
[EUBVP] (Hasher1=0) → lR : (Hasher1'=0);

endmodule
```

Listing 6 Generated PRISM program for the Peer-To-Peer Protocol.

However, for this particular test case, the results of the experiments are not affected, as shown Figure 3 where the results are compared. In this example, since we are comparing the results of two simulations, the two probabilities are slightly different, but it has nothing to do with the model itself.

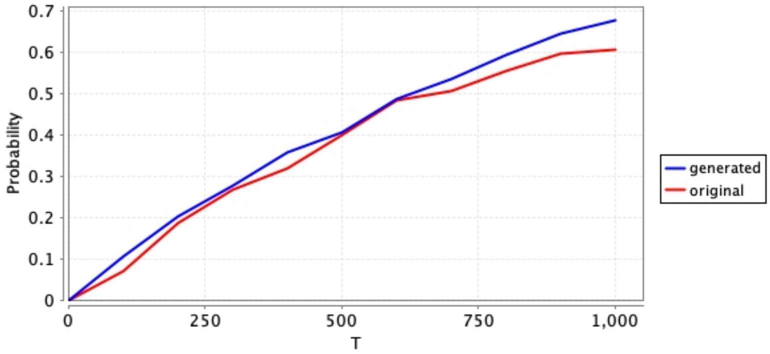
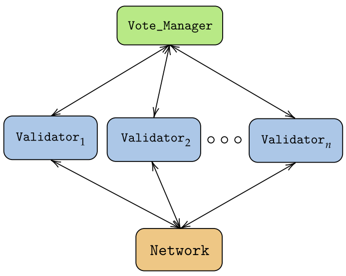


Figure 3 Probability at least one miner has created a block.

2.4 Hybrid Casper Protocol



The last case we study we present is the Hybrid Casper Protocol modelled in PRISM in [4]. The Hybrid Capser protocol is an hybrid blockchain consenus protocol that includes features of the Proof of Work and the Proof of Stake protocols. It was implemented in the Ethereum blockchain [3] as a testing phase before switching to Proof of Stake protocol.

The approach is very similat to the one used for the Proof of Work Bitcoin protocol, so they model Hybrid Casper in PRISM as the parallel composition of n `Validator` modules and the modules `Vote_Manager` and `Network`. The module `Validator` is very similar to the module `Miner` of the previous protocol and the only module that requires an explaination is the `Vote_Manager` that stores the tables containing the votes for each checkpoint and calculates the rewards/penalties.

The modeling language is reported in Listing 7 while (part of) the generated PRISM code can be found in Listing 8.

```
preamble
...
endpreamble
n = 5;
...
{
PoS := Validator[i] -> Validator[i] :
   (+["mR*1"] "(b[i]'=createB(b[i],L[i],c[i]))&(c[i]'=c[i]+1)"&&" " .
  if "!(mod(getHeight(b[i]),EpochSize)=0)"@Validator[i] then{
     Validator[i] -> Network : (["1*1"] "(L[i]'=addBlock(L[i],b[i]))" && foreach(k
         ↪ !=i) "(set[k]'=addBlockSet(set[k],b[i]))"@Network .PoS)
  }
  else{
     Validator[i] -> Network : (["1*1"] "(L[i]'=addBlock(L[i],b[i]))" && foreach(k
         ↪ !=i) "(set[k]'=addBlockSet(set[k],b[i]))"@Network . Validator[i] ->
         ↪ Vote_Manager :(["1*1"] " "&&"(Votes'=addVote(Votes,b[i],stake[i]))".PoS
         ↪ ))
  }
   +["lR*1"] " "&&" " . if "!isEmpty(set[i])"@Validator[i] then {
```

```
      ["1"] "(b[i]'=extractBlock(set[i]))"@Validator[i] .
        if "!canBeInserted(L[i],b[i])"@Validator[i] then {
            PoS
        }
        else{
       if "!(mod(getHeight(b[i]),EpochSize)=0)"@Validator[i] then {
         Validator[i] -> Network : (["1*1"] "(setMiner[i]' = addBlockSet(setMiner[i]
             ↪  , b[i]))"&&"(set[i]' = removeBlock(set[i],b[i]))" . PoS)
       }
       else{
         Validator[i] -> Network : (["1*1"] "(setMiner[i]' = addBlockSet(setMiner[i]
             ↪  , b[i]))"&&"(set[i]' = removeBlock(set[i],b[i]))" . Validator[i] ->
             ↪  Vote_Manager : (["1*1"] " "&&"(Votes'=addVote(Votes,b[i],stake[i]))
             ↪ ".PoS ))
       }
     }
   }
   else{PoS}
    +["rC*1"] "(lastCheck[i]'=extractCheckpoint(listCheckpoints[i],lastCheck[i]))&(
        ↪ heightLast[i]'=getHeight(extractCheckpoint(listCheckpoints[i],lastCheck[i
        ↪ ]))&(votes[i]'=calcVotes(Votes,extractCheckpoint(listCheckpoints[i],
        ↪ lastCheck[i])))"&&" " .
      if "(heightLast[i]=heightCheckpoint[i]+EpochSize)&(votes[i]>=2/3*tot_stake)"
          ↪ @Validator[i] then{
        if "(heightLast[i]=heightCheckpoint[i]+EpochSize)"@Validator[i] then{
          ["1"] "(lastJ[i]'=b[i])&(L[i]'= updateHF(L[i],lastJ[i]))" @Validator[i].
              ↪ Validator[i]->Vote_Manager :(["1*1"]" "&&"(epoch'=height(lastF(L[i
              ↪ ]))&(Stakes'=addVote(Votes,b[i],stake[i]))".PoS)
        }
        else{["1"] "(lastJ[i]'=b[i])"@Validator[i] . PoS}
      }
      else{PoS}
)
}
```

Listing 7 Choreographic language for the Hybrid Casper Protocol.

```
module Validator1
   ...

   [] (Validator1=0) → mR : (b1'=createB(b1,L1,c1))&(c1'=c1+1)&(Validator1'=1);
   [] (Validator1=0) → lR : (Validator1'=2);
   [] (Validator1=0)&(!isEmpty(listCheckpoints1)) →
        rC : (lastCheck1'=extractCheckpoint(listCheckpoints1,lastCheck1))&(
            ↪ heightLast1'=getHeight(extractCheckpoint(listCheckpoints1,lastCheck1
            ↪ )))&(votes1'=calcVotes(Votes,extractCheckpoint(listCheckpoints1,
            ↪ lastCheck1)))&(Validator1'=3);
   [NGRDF] (Validator1=1)&!(mod(getHeight(b1),EpochSize)=0) → 1 : (L1'=addBlock(
        ↪ L1,b1))&(Validator1'=0);
   [] (Validator1=1)&!(!(mod(getHeight(b1),EpochSize)=0)) → 1 : (Validator1'=3);
   [PCRLD] (Validator1=1)&!(mod(getHeight(b1),EpochSize)=0) →
        1 : (L1'=addBlock(L1,b1))&(Validator1'=4);
   [VSJBE] (Validator1=5) → 1 : (Validator1'=0);
   [] (Validator1=2)&!isEmpty(set1) →
```

```
        1 : (b1'=extractBlock(set1))&(Validator1'=4);
    [] (Validator1=4)&!canBeInserted(L1,b1) → (Validator1'=0);
    [] (Validator1=4)&!(!canBeInserted(L1,b1)) → 1 : (Validator1'=6);
    [MDDCF] (Validator1=6)&!(mod(getHeight(b1),EpochSize)=0) →
        1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Validator1'=0);
    [] (Validator1=6)&!(!(mod(getHeight(b1),EpochSize)=0)) → 1 : (Validator1'=8);
    [IQVPA] (Validator1=6)&!(mod(getHeight(b1),EpochSize)=0) →
        1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Validator1'=9);
    [IFNVZ] (Validator1=10) → 1 : (Validator1'=0);
    [] (Validator1=2)&!(!isEmpty(set1)) → 1 : (Validator1'=0);
    [] (Validator1=3)&(heightLast1=heightCheckpoint1+EpochSize)&(votes1>=2/3*
       ↪ tot_stake) → (Validator1'=4);
    [] (Validator1=4)&(heightLast1=heightCheckpoint1+EpochSize) →
        1 : (lastJ1'=b1)&(L1'= updateHF(L1,lastJ1))&(Validator1'=6);
    [EQCYO] (Validator1=6) → 1 : (Validator1'=0);
    [] (Validator1=4)&!((heightLast1=heightCheckpoint1+EpochSize)) →
        1 : (lastJ1'=b1)&(Validator1'=0);
    [] (Validator1=3)&!((heightLast1=heightCheckpoint1+EpochSize)&(votes1>=2/3*
       ↪ tot_stake)) → 1 : (Validator1'=0);
endmodule
...
module Network
   Network : [0..1] init 0;
   set1 : list [];
   set2 : list [];
   set3 : list [];
   set4 : list [];
   set5 : list [];

   [NGRDF] (Network=0) →
        1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,b3))&(set4'=
            ↪ addBlockSet(set4,b4))&(set5'=addBlockSet(set5,b5))&(Network'=0);
   [PCRLD] (Network=0) →
        1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,b3))&(set4'=
            ↪ addBlockSet(set4,b4))&(set5'=addBlockSet(set5,b5))&(Network'=0);
   [MDDCF] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
   [IQVPA] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
   ...
endmodule

module Vote_Manager
   Vote_Manager : [0..1] init 0;
   epoch : [0..10] init 0;
   Votes : hash[];
   tot_stake : [0..120000] init 50;
   stake1 : [0..N] init 10;
   stake2 : [0..N] init 10;
   stake3 : [0..N] init 10;
   stake4 : [0..N] init 10;
   stake5 : [0..N] init 10;

   [VSJBE] (Vote_Manager=0) →
        1 : (Votes'=addVote(Votes,b1,stake1))&(Vote_Manager'=0);
   ...
endmodule
```

Listing 8 Generated PRISM program for the Hybrid Casper Protocol.

The code is very similar to the one presented in [4], the main difference is the fact that our generated model has more lines of code. This is due to the fact that there are some commands that can be merged, but the compiler is not able to do it automatically. This discrepancy between the two models can be observed also in the simulations, reported in Figure 4. Although the results are similar, PRISM takes 39.016 seconds to run the simulations for the generated model, instead of 22.051 seconds needed for the original model.

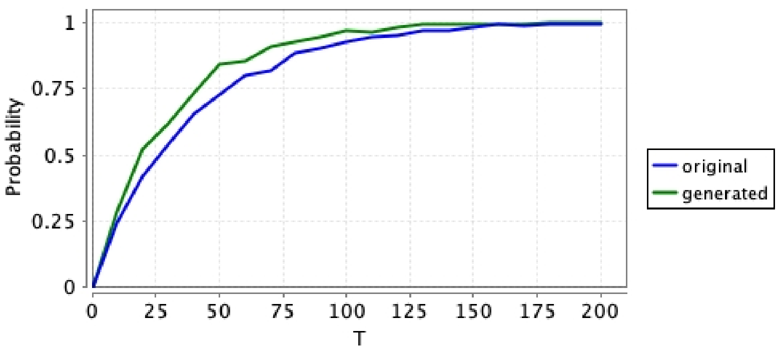


Figure 4 Probability that a block has been created.

2.5 Problems

While testing our choreographic language, we noticed that some of the case studies presented in the PRISM documentation [1] cannot be modeled by using our language. The reasons are various, in this section we try to outline the problems.

- **Asynchronous Leader Election**[5]: processes synchronize with the same label but the conditions are different. We include in our language the `it-then-else` statement but we do not allow the `if-then` (without the `else`). This is done because in this way, we do not incur in deadlock states.
- **Probabilistic Broadcast Protocols**[6]: also in this case, the problem are the labels of the synchronizations. In fact, all the processes synchornize with the same label on every actions. This is not possible in our language, since a label is unique for every synchronization between two (or more) processes.
- **Cyclic Server Polling System**[7]: in this model, the processes $\texttt{station}_i$ do two different things in the same state. More precicely, at the state 0 ($\texttt{s}_i$=0), the processes may synchornize with the process `server` or may change their state without any synchronization. In out language, this cannot be formalized since the synchronization is a branch action, so there should be another option with a synchronization.

[5] `https://www.prismmodelchecker.org/casestudies/asynchronous_leader.php`
[6] `https://www.prismmodelchecker.org/casestudies/prob_broadcast.php`
[7] `https://www.prismmodelchecker.org/casestudies/polling.php`

References

1. Prism documentation. https://www.prismmodelchecker.org/. Accessed: 2023-09-05.
2. Stefano Bistarelli, Rocco De Nicola, Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and Adele Veschetti. Stochastic modeling and analysis of the bitcoin protocol in the presence of block communication delays. *Concurr. Comput. Pract. Exp.*, 35(16), 2023. doi:10.1002/cpe.6749.
3. Vitalik Buterin. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
4. Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and Adele Veschetti. Resilience of hybrid casper under varying values of parameters. *Distributed Ledger Technol. Res. Pract.*, 2(1):5:1–5:25, 2023. doi:10.1145/3571587.
5. D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.
6. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.