

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

In this section, we provide the formal definition of our choreographic language as well as process algebra representing PRISM [?].

1.1 PRISM

We start by describing PRISM semantics. Except from transforming some informal text in precise rules, Our formalisation closely follows that found on the PRISM website [?].

Syntax. Let \mathbf{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 \mathbf{Var} , and v over a (possibly infinite) set of values \mathbf{Val} . Then, the syntax of PRISM is given by the following grammar:

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

Networks are the top syntactic category for system of modules composed together. The term $\mathbf{0}$ represent an empty network. A module $\mathbf{p} : \{F_i\}_i$ is identified by its name \mathbf{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?](#)



© God;
licensed under Creative Commons License CC-BY 4.0

International Conference on Blah.

Editors: John Q. Open and Joan R. Access; Article No. m; pp. m:1–m:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

30 **Semantics.** In order to give a probabilistic semantics to PRISM, we proceed by steps. First,
 31 we define $\llbracket - \rrbracket$, as the closure of the following rules:

$$\begin{array}{c}
 \frac{}{F_i \in \llbracket \mathbf{p} : \{F_i\}_i \rrbracket} \text{ (Module)} \quad \frac{\llbracket E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_j \rrbracket \quad j \in \{1, 2\}}{\llbracket E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_1 | [A] | M_2 \rrbracket} \text{ (Par}_1\text{)} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_j \rrbracket \quad a \notin A \quad j \in \{1, 2\}}{[a]E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_1 | [A] | M_2 \rrbracket} \text{ (Par}_2\text{)} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_1 \rrbracket \quad [a]E' \rightarrow \{\lambda'_j : y_j = E'_j\}_{j \in J} \in \llbracket M_2 \rrbracket \quad a \in A}{\llbracket E \wedge E' \rightarrow \{\lambda_i * \lambda'_j : x_i = E_i \wedge y_j = E'_j\}_{i \in I, j \in J} \in \llbracket M_1 | [A] | M_2 \rrbracket} \text{ (Par}_3\text{)} \\
 \\
 \frac{\llbracket E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket}{\llbracket E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M/A \rrbracket} \text{ (Hide}_1\text{)} \quad \frac{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket \quad a \notin A}{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M/A \rrbracket} \text{ (Hide}_2\text{)} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket \quad a \in A}{\llbracket E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M/A \rrbracket} \text{ (Hide}_3\text{)} \quad \frac{\llbracket E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket}{\llbracket E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket \sigma M \rrbracket} \text{ (Subst}_1\text{)} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket \quad a \notin \text{dom}(\sigma)}{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket \sigma M \rrbracket} \text{ (Subst}_2\text{)} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket M \rrbracket \quad a \in \text{dom}(\sigma)}{[\sigma a]E \rightarrow \{\lambda_j : x_i = E_i\}_{i \in I} \in \llbracket \sigma M \rrbracket} \text{ (Subst}_3\text{)}
 \end{array}$$

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

38 Now, given $\llbracket N \rrbracket$, we define a transition system that shows how the system evolves. In
 39 order to do so, let **state** be a function that given a variable in **Var** returns a value in **Val**.
 40 Then, given an initial state state_0 , we can define a transition system where each of node is a
 41 (different) **state** function. Then, we can move from state_1 to state_2 whenever

42 That means that ones we have a set of executable rules, we can start building a transition
 43 system. In order to do so, we

$$W(M) = \{F \mid F \in \llbracket M \rrbracket\}$$

44 $X = \{x_1, \dots, x_n\}$

$$\sigma : X \rightarrow V$$

1.2 Choreographies

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

$$(\text{Chor}) \quad C ::= \{p_i\}_{i \in I} \Sigma \{\lambda_j : x_j = E_j; C_j\}_{j \in J} \mid \text{if } E @ \{p_i\}_{i \in I} \text{ then } C_1 \text{ else } C_2 \mid X \mid 0$$

We briefly comment the various constructs. The syntactic category C denotes choreographic programmes. The term $\{p_i\}_{i \in I} + \{\lambda_j : x_j = E_j; C_j\}_{j \in J}$ denotes an interaction between the roles p_i . The value λ_j is a real number representing the rate. ...

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.

$$f : C \times \text{network} \times \text{States} \longrightarrow \text{network} \quad \text{network} : \mathcal{R} \longrightarrow \text{Set}(F)$$

$$f\left(p_1 \longrightarrow \{p_i\}_{i \in I} + \{\lambda_j : x_j = E_j; C_j\}_{j \in J}, \text{network}, \bar{s}\right)$$

=

ℓ fresh;

for $p_k \in \text{roles}\{$

for $j \in J\{$

$\text{network} = \text{add}(p_k, [\ell] \ s_{p_k} = s_k \rightarrow \Sigma_j \lambda_j : x_j = E_j \ \& \ s'_{p_k} = s_k + 1);$

$\}$

$\}$

for $j \in J\{$

$\text{network} = f(C_j, \text{network}, \bar{s}');$

$\}$

return network

where $\bar{s} = (s_1, \dots, s_n)$ and $\bar{s}' = (s_1 + 1, \dots, s_n + 1)$ for $n = |\text{roles}|$.

$$f\left(\text{if } E @ \{p_i\}_{i \in I} \text{ then } C_1 \text{ else } C_2, \text{network}, \bar{s}\right)$$

=

for $p_k \in \text{roles}\{$

$\text{network} = \text{add}(p_k, [\] \ s_{p_k} = s_k \ \& \ f(E)) + f(C_1, \text{network}, \bar{s});$

$\text{network} = \text{add}(p_k, [\] \ s_{p_k} = s_k \ \& \ !f(E)) + f(C_2, \text{network}, \bar{s});$

$\}$

return network

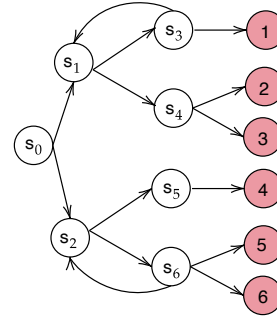
2 Tests

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 repository¹; 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 Program²[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.



```

74 preamble
75 "dtmc"
76 endpreamble
77
78
79 n = 1;
80
81 Dice → Dice : "d : [0..6] init 0;" ;
82
83 {
84 DiceProtocol0 := Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol1
85                               +["0.5*1"] " "&&" " . DiceProtocol2)
86
87 DiceProtocol1 := Dice → Dice : (+["0.5*1"] " "&&" " .
88                               Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol1
89                               +["0.5*1"] "(d'=1)"&&" " . DiceProtocol3)
90                               +["0.5*1"] " "&&" " .
91                               Dice → Dice : (+["0.5*1"] "(d'=2)"&&" " . DiceProtocol3
92                               +["0.5*1"] "(d'=3)"&&" " . DiceProtocol3)
93
94 DiceProtocol2 := Dice → Dice : (+["0.5*1"] " "&&" " .
95                               Dice → Dice : (+["0.5*1"] " "&&" " . DiceProtocol2
96                               +["0.5*1"] "(d'=4)"&&" " . DiceProtocol3)
97                               +["0.5*1"] " "&&" " .
98                               Dice → Dice : (+["0.5*1"] "(d'=5)"&&" " . DiceProtocol3
99                               +["0.5*1"] "(d'=6)"&&" " . DiceProtocol3)
100
101 DiceProtocol3 := Dice → Dice : ([ "1*1" ] " "&&" " . DiceProtocol3)
102 }

```

¹ <https://www.prismmodelchecker.org/casestudies/>

² <https://www.prismmodelchecker.org/casestudies/dice.php>

■ **Listing 1** Choreographic language for the Dice Program.

```

104 dtmc
105
106
107 module Dice
108     Dice : [0..11] init 0;
109     d : [0..6] init 0;
110
111     [] (Dice=0) → 0.5 : (Dice'=2) + 0.5 : (Dice'=6);
112     [] (Dice=2) → 0.5 : (Dice'=3) + 0.5 : (Dice'=4);
113     [] (Dice=3) → 0.5 : (Dice'=2) + 0.5 : (d'=1)&(Dice'=10);
114     [] (Dice=4) → 0.5 : (d'=2)&(Dice'=10) + 0.5 : (d'=3)&(Dice'=10);
115     [] (Dice=6) → 0.5 : (Dice'=7) + 0.5 : (Dice'=8);
116     [] (Dice=7) → 0.5 : (Dice'=6) + 0.5 : (d'=4)&(Dice'=10);
117     [] (Dice=8) → 0.5 : (d'=5)&(Dice'=10) + 0.5 : (d'=6)&(Dice'=10);
118     [] (Dice=10) → 1 : (Dice'=10);
119
120 endmodule
121

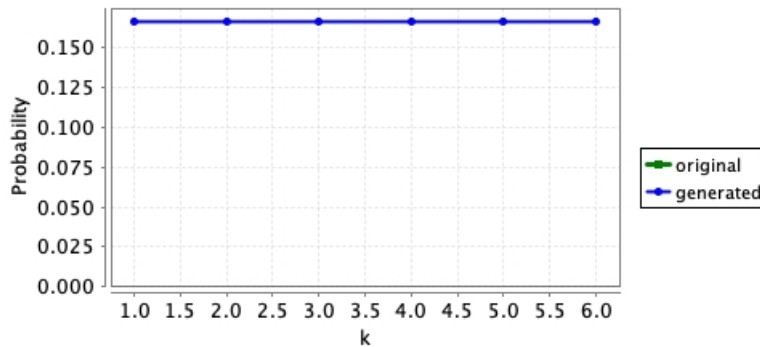
```

■ **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

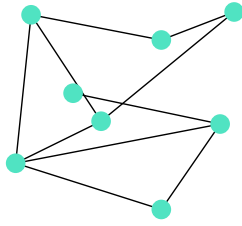
$$d=k \text{ for } k = 1, \dots, 6 \text{ is } 1/6.$$

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 equivalent.



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

2.2 Random Graphs Protocol



The second case study we report is the random graphs protocol presented in the PRISM documentation³. It investigates the likelihood that a pair of nodes are connected in a random graph. More precisely, we take into account 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 repository⁴).

```

140 preamble
141 "mdp"
142 "const double p;"
143 endpreamble
144
145 n = 3;
146
147 PC -> PC : " ";
148 M[i] -> i in [1..n] M[i] : "varM[i] : bool;";
149 P[i] -> i in [1..n] P[i] : "varP[i] : bool;";
150
151 {
152   GraphConnected0 :=
153     PC -> M[i] : (+["1*p"] " " "&&"(varM[i]')==true)". END
154               +["1*(1-p)" " " "&&"(varM[i]')==false)". END)
155     PC -> P[i] : (+["1*p"] " " "&&"(varP[i]')==true)". END
156               +["1*(1-p)" " " "&&"(varP[i]')==false)".
157               if "(PC=6)&!varP[i]&((varP[i] & varM[i]) | (varM[i+1] & varP[
158                 ↪ i+2]))" "@P[i] then {
159                 ["1"] (varP[i]')==true)"@P[i] . GraphConnected0
160               })
161   }
162 }
163

```

■ Listing 3 Choreographic language for the Random Graphs Protocol.

```

164 mdp
165 const double p;
166
167 module PC
168   PC : [0..7] init 0;
169
170

```

³ https://www.prismmodelchecker.org/casestudies/graph_connected.php

⁴ <https://github.com/adeleveschetti/choreography-to-PRISM>

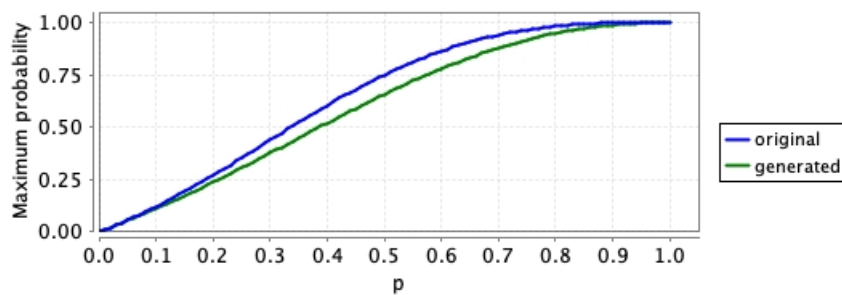
```

171 [DPPGR] (PC=0) → 1 : (PC'=1);
172 [YCJJG] (PC=1) → 1 : (PC'=2);
173 [TWGVA] (PC=2) → 1 : (PC'=3);
174 [NODPZ] (PC=3) → 1 : (PC'=4);
175 [FDALJ] (PC=4) → 1 : (PC'=5);
176 [DCKXC] (PC=5) → 1 : (PC'=6);
177 endmodule
178
179 module M1
180   M1 : [0..1] init 0;
181   varM1 : bool;
182
183   [DPPGR] (M1=0) → p : (varM1'=true)&(M1'=0) + (1-p) : (varM1'=false)&(M1'=0);
184 endmodule
185
186 ...
187
188 module P1
189   P1 : [0..3] init 0;
190   varP1 : bool;
191
192   [NODPZ] (P1=0) → p : (varP1'=true)&(P1'=0) + (1-p) : (varP1'=false)&(P1'=0);
193   [] (P1=0)&(PC=6)&!varP1&((varP1 & varM1) | (varM2& varP3))
194     → 1 : (varP1'=true)&(P1'=0);
195 endmodule
196 ...
197

```

■ **Listing 4** Generated PRISM program for the Random Graphs Protocol.

198 The model is very similar to the one presented in the PRISM repository, the main
 199 difference is that we use state variables also for the modules P_i and M_i , where in the original
 200 model they were not required. However, this does not affect the behaviour of the model, as
 201 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.

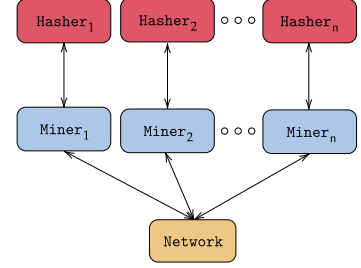
202

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.

```

218 preamble
219 ...
220 endpreamble
221
222 n = 4;
223
224 ...
225
226 {
227   PoW := Hasher[i] -> Miner[i] :
228     (+["mR*hr[i]" " "&&"(b[i]'=createB(b[i],B[i],c[i]))&(c[i]'=c[i]+1)" " .
229       Miner[i] -> Network :
230         ([ "rB*1" " "(B[i]'=addBlock(B[i],b[i]))" "&&
231           foreach(k != i) "(set[k]'=addBlockSet(set[k],b[i]))" @Network .PoW
232         +["lR*hr[i]" " "&&" " " .
233         if "!isEmpty(set[i])"@Miner[i] then {
234           ["r" " "(b[i]'=extractBlock(set[i]))"@Miner[i] .
235           Miner[i] -> Network :
236             ([ "1*1" " "(setMiner[i]' = addBlockSet(setMiner[i] , b[i]))"&&
237             ↪ "(set[i]' = removeBlock(set[i],b[i]))" . PoW
238         }
239       else{
240         if "canBeInserted(B[i],b[i])"@Miner[i] then {
241           ["1" " "(B[i]'=addBlock(B[i],b[i]))&&(setMiner[i]'=removeBlock
242             ↪ (setMiner[i],b[i]))"@Miner[i] . Pow
243         }
244       else{
245         PoW
246       }
247     }
248   )
249 }
250
251

```

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

Part of the generated PRISM code is shown in Listing 6, the modules *Miner₂*, *Miner₃*, *Miner₄* and *Hasher₂*, *Hasher₃*, *Hasher₄* are equivalent to *Miner₁* and *Hasher₁*, 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

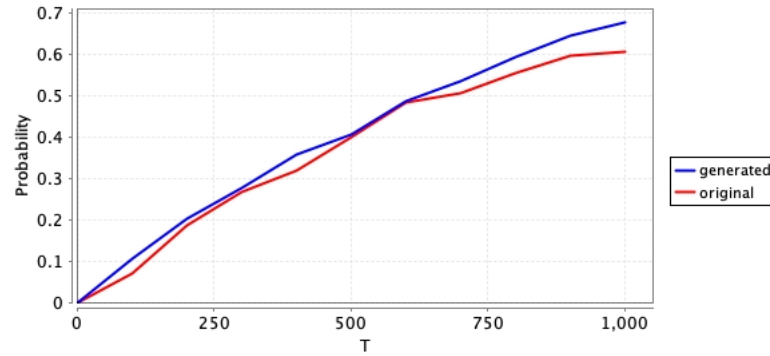
```

257 ...
258 ...
259 ...
260 module Miner1
261   Miner1 : [0..7] init 0;
262   b1 : block {m1,0;genesis,0} ;
263   B1 : blockchain [{genesis,0;genesis,0}];
264   c1 : [0..N] init 0;
265   setMiner1 : list [];
266
267   [PZKYT] (Miner1=0) → hR1 : (b1'=createB(b1,B1,c1))&(c1'=c1+1)&(Miner1'=1);
268   [EUBVP] (Miner1=0) → hR1 : (Miner1'=2);
269   [HXYKO] (Miner1=1) → 1 : (B1'=addBlock(B1,b1))&(Miner1'=0);
270   [] (Miner1=2)&!isEmpty(set1) → r : (b1'=extractBlock(set1))&(Miner1'=4);
271   [SRKSV] (Miner1=4) → 1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Miner1'=0)
272   ↪ ;
273   [] (Miner1=2)&!(!isEmpty(set1)) → 1 : (Miner1'=5);
274   [] (Miner1=5)&canBeInserted(B1,b1) → 1 : (B1'=addBlock(B1,b1))&(setMiner1'=
275   ↪ removeBlock(setMiner1,b1))&(Miner1'=0);
276   [] (Miner1=5)&!(!canBeInserted(B1,b1)) → 1 : (Miner1'=0);
277
278 endmodule
279 ...
280 module Network
281   Network : [0..1] init 0;
282   set1 : list [];
283   ...
284
285   [HXYKO] (Network=0) → 1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,
286   ↪ b3))&(set4'=addBlockSet(set4,b4))&(Network'=0);
287   [SRKSV] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
288   ...
289
290 endmodule
291
292 module Hasher1
293   Hasher1 : [0..1] init 0;
294
295   [PZKYT] (Hasher1=0) → mR : (Hasher1'=0);
296   [EUBVP] (Hasher1=0) → lR : (Hasher1'=0);
297
298 endmodule
299

```

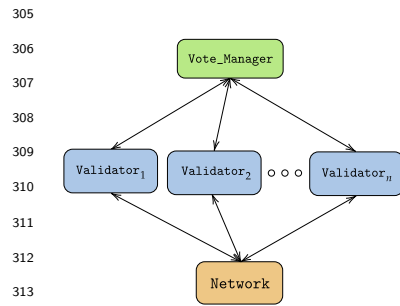
■ **Listing 6** Generated PRISM program for the Peer-To-Peer Protocol.

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.

304 2.4 Hybrid Casper Protocol



The last case we study we present is the Hybrid Casper Protocol modelled in PRISM in [4]. The Hybrid Casper protocol is an hybrid blockchain consensus 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 similar 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 explanation 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.

```

320 preamble
321 ...
322 endpreamble
323 n = 5;
324 ...
325 {
326   PoS := Validator[i] -> Validator[i] :
327     (+["mR*1"] "(b[i]'=createB(b[i],L[i],c[i]))&(c[i]'=c[i]+1)"&&" " .
328     if "!(mod(getHeight(b[i]),EpochSize)=0)"@Validator[i] then{
329       Validator[i] -> Network : ([["1*1"] "(L[i]'=addBlock(L[i],b[i]))" && foreach(k
330         ↪ !=i) "(set[k]'=addBlockSet(set[k],b[i]))"@Network .PoS)
331     }
332   else{
333     Validator[i] -> Network : ([["1*1"] "(L[i]'=addBlock(L[i],b[i]))" && foreach(k
334       ↪ !=i) "(set[k]'=addBlockSet(set[k],b[i]))"@Network . Validator[i] ->
335       ↪ Vote_Manager : ([["1*1"] " "&&"(Votes'=addVote(Votes,b[i],stake[i]))".PoS
336       ↪ ))
337   }
338 }
339 +["lR*1"] " "&&" " . if "isEmpty(set[i])"@Validator[i] then {

```

```

340   ["1"] "(b[i]'=extractBlock(set[i]))"@Validator[i] .
341   if "!(canBeInserted(L[i],b[i]))"@Validator[i] then {
342       PoS
343   }
344   else{
345   if "!(mod(getHeight(b[i]),EpochSize)=0)"@Validator[i] then {
346       Validator[i] -> Network : ([ "1*1" ] "(setMiner[i]' = addBlockSet(setMiner[i]
347       ↪ , b[i]))"&&"(set[i]' = removeBlock(set[i],b[i]))" . PoS)
348   }
349   else{
350       Validator[i] -> Network : ([ "1*1" ] "(setMiner[i]' = addBlockSet(setMiner[i]
351       ↪ , b[i]))"&&"(set[i]' = removeBlock(set[i],b[i]))" . Validator[i] ->
352       ↪ Vote_Manager : ([ "1*1" ] " "&&"(Votes'=addVote(Votes,b[i],stake[i]))
353       ↪ ".PoS ))
354   }
355   }
356   }
357   else{PoS}
358   +["rC*1"] "(lastCheck[i]'=extractCheckpoint(listCheckpoints[i],lastCheck[i]))&(
359   ↪ heightLast[i]'=getHeight(extractCheckpoint(listCheckpoints[i],lastCheck[i]
360   ↪ ))&(votes[i]'=calcVotes(Votes,extractCheckpoint(listCheckpoints[i],
361   ↪ lastCheck[i])))"&&" " .
362   if "(heightLast[i]=heightCheckpoint[i]+EpochSize)&(votes[i]>=2/3*tot_stake)"
363   ↪ @Validator[i] then{
364       if "(heightLast[i]=heightCheckpoint[i]+EpochSize)"@Validator[i] then{
365           ["1"] "(lastJ[i]'=b[i])&(L[i]'= updateHF(L[i],lastJ[i]))" @Validator[i].
366           ↪ Validator[i]->Vote_Manager :([ "1*1" ] " "&&"(epoch'=height(lastF(L[i]
367           ↪ ))&(Stakes'=addVote(Votes,b[i],stake[i]))".PoS)
368       }
369       else{["1"] "(lastJ[i]'=b[i])"@Validator[i] . PoS}
370   }
371   else{PoS}
372   )
373   }
374

```

■ Listing 7 Choreographic language for the Hybrid Casper Protocol.

```

375 module Validator1
376 ...
377
378
379 [] (Validator1=0) → mR : (b1'=createB(b1,L1,c1))&(c1'=c1+1)&(Validator1'=1);
380 [] (Validator1=0) → lR : (Validator1'=2);
381 [] (Validator1=0)&(!isEmpty(listCheckpoints1)) →
382     rC : (lastCheck1'=extractCheckpoint(listCheckpoints1,lastCheck1))&(
383     ↪ heightLast1'=getHeight(extractCheckpoint(listCheckpoints1,lastCheck1
384     ↪ ))&(votes1'=calcVotes(Votes,extractCheckpoint(listCheckpoints1,
385     ↪ lastCheck1)))&(Validator1'=3);
386 [NGRDF] (Validator1=1)&!(mod(getHeight(b1),EpochSize)=0) → 1 : (L1'=addBlock(
387     ↪ L1,b1))&(Validator1'=0);
388 [] (Validator1=1)&(!(mod(getHeight(b1),EpochSize)=0)) → 1 : (Validator1'=3);
389 [PCRLD] (Validator1=1)&!(mod(getHeight(b1),EpochSize)=0) →
390     1 : (L1'=addBlock(L1,b1))&(Validator1'=4);
391 [VSJBE] (Validator1=5) → 1 : (Validator1'=0);
392 [] (Validator1=2)&(!isEmpty(set1)) →

```

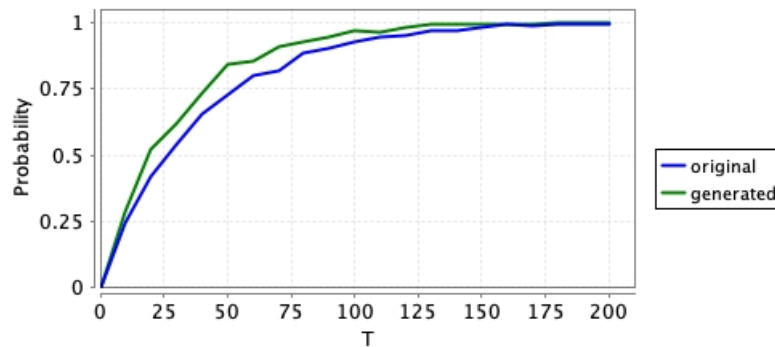
```

393     1 : (b1'=extractBlock(set1)) & (Validator1'=4);
394 [] (Validator1=4) & (!canBeInserted(L1,b1)) → (Validator1'=0);
395 [] (Validator1=4) & (!(!canBeInserted(L1,b1))) → 1 : (Validator1'=6);
396 [MDDCF] (Validator1=6) & !(mod(getHeight(b1),EpochSize)=0) →
397     1 : (setMiner1' = addBlockSet(setMiner1 , b1)) & (Validator1'=0);
398 [] (Validator1=6) & !(mod(getHeight(b1),EpochSize)=0) → 1 : (Validator1'=8);
399 [IQVPA] (Validator1=6) & !(mod(getHeight(b1),EpochSize)=0) →
400     1 : (setMiner1' = addBlockSet(setMiner1 , b1)) & (Validator1'=9);
401 [IFNVZ] (Validator1=10) → 1 : (Validator1'=0);
402 [] (Validator1=2) & (!isEmpty(set1)) → 1 : (Validator1'=0);
403 [] (Validator1=3) & (heightLast1=heightCheckpoint1+EpochSize) & (votes1>=2/3*
404     ↪ tot_stake) → (Validator1'=4);
405 [] (Validator1=4) & (heightLast1=heightCheckpoint1+EpochSize) →
406     1 : (lastJ1'=b1) & (L1'= updateHF(L1,lastJ1)) & (Validator1'=6);
407 [EQCYO] (Validator1=6) → 1 : (Validator1'=0);
408 [] (Validator1=4) & !(heightLast1=heightCheckpoint1+EpochSize) →
409     1 : (lastJ1'=b1) & (Validator1'=0);
410 [] (Validator1=3) & !(heightLast1=heightCheckpoint1+EpochSize) & (votes1>=2/3*
411     ↪ tot_stake) → 1 : (Validator1'=0);
412 endmodule
413 ...
414 module Network
415     Network : [0..1] init 0;
416     set1 : list [];
417     set2 : list [];
418     set3 : list [];
419     set4 : list [];
420     set5 : list [];
421
422     [NGRDF] (Network=0) →
423         1 : (set2'=addBlockSet(set2,b2)) & (set3'=addBlockSet(set3,b3)) & (set4'=
424             ↪ addBlockSet(set4,b4)) & (set5'=addBlockSet(set5,b5)) & (Network'=0);
425     [PCRLD] (Network=0) →
426         1 : (set2'=addBlockSet(set2,b2)) & (set3'=addBlockSet(set3,b3)) & (set4'=
427             ↪ addBlockSet(set4,b4)) & (set5'=addBlockSet(set5,b5)) & (Network'=0);
428     [MDDCF] (Network=0) → 1 : (set1' = removeBlock(set1,b1)) & (Network'=0);
429     [IQVPA] (Network=0) → 1 : (set1' = removeBlock(set1,b1)) & (Network'=0);
430     ...
431 endmodule
432
433 module Vote_Manager
434     Vote_Manager : [0..1] init 0;
435     epoch : [0..10] init 0;
436     Votes : hash[];
437     tot_stake : [0..120000] init 50;
438     stake1 : [0..N] init 10;
439     stake2 : [0..N] init 10;
440     stake3 : [0..N] init 10;
441     stake4 : [0..N] init 10;
442     stake5 : [0..N] init 10;
443
444     [VSJBE] (Vote_Manager=0) →
445         1 : (Votes'=addVote(Votes,b1,stake1)) & (Vote_Manager'=0);
446     ...
447 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 synchronizes 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 **station_i** do two different things in the same state. More precisely, at the state 0 (**s_i=0**), the processes may synchronize with the process **server** or may change their state without any synchronization. In our 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>

472 ——— **References** ———

- 473 1 Prism documentation. <https://www.prismmodelchecker.org/>. Accessed: 2023-09-05.
- 474 2 Stefano Bistarelli, Rocco De Nicola, Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and
475 Adele Veschetti. Stochastic modeling and analysis of the bitcoin protocol in the presence of block
476 communication delays. *Concurr. Comput. Pract. Exp.*, 35(16), 2023. doi:10.1002/cpe.6749.
- 477 3 Vitalik Buterin. Ethereum white paper. [https://github.com/ethereum/wiki/wiki/](https://github.com/ethereum/wiki/wiki/White-Paper)
478 [White-Paper](https://github.com/ethereum/wiki/wiki/White-Paper), 2013.
- 479 4 Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and Adele Veschetti. Resilience of hybrid
480 casper under varying values of parameters. *Distributed Ledger Technol. Res. Pract.*, 2(1):5:1–
481 5:25, 2023. doi:10.1145/3571587.
- 482 5 D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter
483 The complexity of nonuniform random number generation. Academic Press, 1976.
- 484 6 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [https://bitcoin.org/](https://bitcoin.org/bitcoin.pdf)
485 [bitcoin.pdf](https://bitcoin.org/bitcoin.pdf), 2008.