

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 Language

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

1.1 Choreographies

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

$$\begin{aligned} \text{(Chor)} \quad C &::= \{p_i\}_{i \in I} + \{\lambda_j : x_j = E_j; C_j\}_{j \in J} \mid \text{if } E @ p \text{ then } C_1 \text{ else } C_2 \mid X \mid \mathbf{0} \\ \text{(Expr)} \quad E &::= f(\tilde{E}) \mid x \mid v \\ \text{(Rates)} \quad \lambda &\in \mathbb{R} \quad \text{(Variables)} \quad x \in \mathbf{Var} \quad \text{(Values)} \quad v \in \mathbf{Val} \end{aligned}$$

We briefly comment the various constructs. The syntactic category C denotes choreographic programmes. The term $p \longrightarrow \{p_i\}_{i \in I} \oplus \{\lambda_j x_j = E_j : C_j\}_{j \in J}$ denotes an interaction between roles $p_i \dots$

1.2 PRISM

Syntax.

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

Semantics. We construct all the enables commands by applying a closure to the following



© 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:12

Leibniz International Proceedings in Informatics



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

24 rules.

$$\begin{array}{c}
 \frac{\llbracket E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_j \rrbracket \quad j \in \{1, 2\}}{\llbracket E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_1 \mid [A] \mid M_2 \rrbracket} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_j \rrbracket \quad a \notin A \quad j \in \{1, 2\}}{[a]E \rightarrow \{\lambda_i : x_i = E_i\}_{i \in I} \in \llbracket M_1 \mid [A] \mid M_2 \rrbracket} \\
 \\
 \frac{[a]E \rightarrow \{\lambda_j : x_j = E_j\}_{j \in I} \in \llbracket M_1 \rrbracket \quad [a]E' \rightarrow \{\lambda_j : x'_j = E'_j\}_{j \in J} \in \llbracket M_2 \rrbracket \quad a \in A}{[a]E \wedge E' \rightarrow \{\lambda_i * \lambda'_j : x_i = E_i \wedge x'_j = E'_j\}_{i \in I, j \in J} \in \llbracket M_1 \mid [A] \mid M_2 \rrbracket}
 \end{array}$$

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

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

$$28 \quad X = \{x_1, \dots, x_n\}$$

$$\sigma : X \rightarrow V$$

29 1.3 Projection from Choreographies to PRISM

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

$$32 \quad f : C \rightarrow \mathcal{R} \mapsto F$$

$$\begin{aligned}
 & f(\mathbf{p}_1 \longrightarrow \{\mathbf{p}_i\}_{i \in I} \oplus \{\lambda_j\}x_j = E_j : C_j\}_{j \in J}) = \\
 & = \begin{cases} \left([\lambda_{j_1}]x_{j_1} = f(E_{j_1}) \right)_{\mathbf{p}} \cdot f(\oplus \{\lambda_j\}x_j = E_j : C_j\}_{j \in J \setminus \{j_1\}}) \cdot f(C_{j_1}) & \text{if } \mathbf{p} = \mathbf{p}_1 \vee \mathbf{p} \in \{\mathbf{p}_i\}_{i \in I} \\ f(C_j) & \text{if } \mathbf{p} \neq \mathbf{p}_1 \wedge \mathbf{p} \notin \{\mathbf{p}_i\}_{i \in I} \end{cases} \\
 & f(\text{if } E @ \mathbf{p} \text{ then } C_1 \text{ else } C_2) = \begin{cases} f(E) \cdot f(C_1) \cdot f(C_2) & \text{if } \mathbf{p} \in \mathbf{roles} \\ \perp & \text{otherwise} \end{cases} \\
 & f(X) = ?? \\
 & f(\mathbf{0}) = \perp
 \end{aligned}$$

$$34 \quad f : [C_1, \dots, C_n] \rightarrow \text{String} \mapsto \text{String}$$

$$35 \quad \text{CASE 1: } \mathbf{C_i} \equiv \mathbf{p}_1 \longrightarrow \{\mathbf{p}_i\}_{i \in I} \oplus \{\lambda_j\}x_j = E_j : C_j\}_{j \in J}$$

```

36
37
38   f([Ci, ..., Cn], code) :
39     label = generateNewLabel()
40     for cj in Ci :
41       for p ∈ roles(Ci):
42         newCode = "[label] (xj = Ej)p"
43         code = code + newCode
44     f([Ci+1, ..., Cn], code)
45

```

46 where $\mathbf{roles}(C_i) := \mathbf{p}_1 \cup \{\mathbf{p}_i\}_{i \in I}$.

```
47
48 CASE 2:  $C_i \equiv \text{if } E@p \text{ then } C_1 \text{ else } C_2$ 
49
50 f( $[C_i, \dots, C_n]$ , code) :
51   code = code +  $(E)_p$ 
52   f( $C_1$ , code)
53   f( $C_2$ , code)
54   f( $[C_{i+1}, \dots, C_n]$ , code)
55
```

m:4 A Choreographic Language for PRISM

```

56
57
58   network :  $\mathcal{R} \longrightarrow \text{Set}(F)$ 
59
60   f( $C_1, \dots, C_n, \text{network}$ ) =
61
62   CASE 1:  $\forall i. C_i \equiv p_1 \longrightarrow \{p_i\}_{i \in I} \oplus \{[\lambda_j]x_j = E_j : D_j\}_{j \in J}$ 
63
64   ->
65
66       label = generateNewLabel()
67
68
69
70
71   for  $c_j$  in  $C_i$  :
72   for  $p \in \text{roles}(C_i)$ :
73   newCode = "[label] ( $x_j = E_j$ )p"
74   code = code + newCode
75   f( $[C_{i+1}, \dots, C_n], \text{code}$ )
76

```

$$f\left(p_1 \longrightarrow \{p_2\} \oplus \left\{ \begin{array}{l} [\lambda_1]x = 5 : p_1 \longrightarrow \{p_2\} \oplus \{[\lambda_3]y = 5\} \\ [\lambda_2]y = 10 : p_1 \longrightarrow \{p_2\} \oplus \{[\lambda_4]x = 10\} \end{array} \right\}, p_1 : \emptyset \parallel p_2 : \emptyset \right)$$

$$=$$

```

77   label = newlabel();
78   for  $p_i$ {
       add( $p_i, [\text{label}]s_{p_i} = \text{state}(p_i) \rightarrow \left\{ \begin{array}{l} \lambda_1 : x' = 5; \text{state}(p_i)' = \text{generatenewstate}(p_i) \\ \lambda_2 : y' = 10; \text{state}(p_i)' = \text{generatenewstate}(p_i) \end{array} \right\}$ 
       f( $p_1 \longrightarrow \{p_2\} \oplus \{[\lambda_3]y = 5\}, \text{network}'$ ) =  $\text{network}''$ 
       return f( $p_1 \longrightarrow \{p_2\} \oplus \{[\lambda_4]x = 10\}, \text{network}''$ )

```

```

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

```

$$f\left(p_1 \longrightarrow \{p_i\}_{i \in I} \oplus \{[\lambda_j]x_j = E_j : D_j\}_{j \in J}, \text{network} \right)$$

$$=$$

```

       label = newlabel();
       for  $p_k \in \text{roles}$ {
           for  $j \in J$ {
79               network = add( $p_k, [\text{label}]s_{p_k} = \text{state}(p_k) \rightarrow \lambda_j : x_j = E_j \ \& \ s'_{p_k} = \text{genNewState}(p_k)$ );
           }
       }
       for  $j \in J$ {
           network = f( $D_j, \text{network}$ );
       }
       return network

```

$$f\left(\text{if } E@p \text{ then } C_1 \text{ else } C_2, \text{network}\right)$$
$$=$$

80

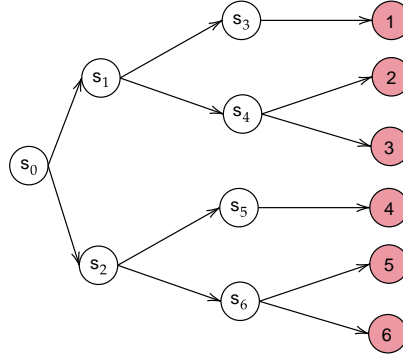
```
network = add(p, []sp = state(p) & f(E));  
network = f(C1, network);  
network = f(C2, network);  
return network
```

2 Tests

We tested our language by various examples.

2.1 The Dice Program

The first example we present is the Dice Program¹. The following program models a die using only fair coins. Starting at the root vertex (state 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.



We modelled the program using the choreographic language (Listing 1) and we were able to generate the corresponding PRISM program, reported in Listing 2.

```

90 preamble
91 "dtmc"
92 endpreamble
93
94 n = 1;
95 Dice → Dice : "d : [0..6] init 0;" ;
96
97 {
98 DiceProtocol0 := Dice → Dice : (+["0.5*1" " "&&" " . DiceProtocol1
99   +["0.5*1" " "&&" " . DiceProtocol2)
100
101 DiceProtocol1 := Dice → Dice : (+["0.5*1" " "&&" " .
102   Dice → Dice : (+["0.5*1" " "&&" " . DiceProtocol1
103     +["0.5*1" " "(d'=1)"&&" " . DiceProtocol3)
104     +["0.5*1" " "&&" " .
105   Dice → Dice : (+["0.5*1" " "(d'=2)"&&" " . DiceProtocol3
106     +["0.5*1" " "(d'=3)"&&" " . DiceProtocol3)
107
108 DiceProtocol2 := Dice → Dice : (+["0.5*1" " "&&" " .
109   Dice → Dice : (+["0.5*1" " "&&" " . DiceProtocol2
110     +["0.5*1" " "(d'=4)"&&" " . DiceProtocol3)
111     +["0.5*1" " "&&" " .
112   Dice → Dice : (+["0.5*1" " "(d'=5)"&&" " . DiceProtocol3
113     +["0.5*1" " "(d'=6)"&&" " . DiceProtocol3)
114

```

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

```

115
116 DiceProtocol3 := Dice → Dice : ([ "1*1" " "&" ".DiceProtocol3)
117 }
118

```

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

```

119
120 dtmc
121
122 module Dice
123     Dice : [0..11] init 0;
124     d : [0..6] init 0;
125
126     [] (Dice=0) → 0.5 : (Dice'=2) + 0.5 : (Dice'=6);
127     [] (Dice=2) → 0.5 : (Dice'=3) + 0.5 : (Dice'=4);
128     [] (Dice=3) → 0.5 : (Dice'=2) + 0.5 : (d'=1)&(Dice'=10);
129     [] (Dice=4) → 0.5 : (d'=2)&(Dice'=10) + 0.5 : (d'=3)&(Dice'=10);
130     [] (Dice=6) → 0.5 : (Dice'=7) + 0.5 : (Dice'=8);
131     [] (Dice=7) → 0.5 : (Dice'=6) + 0.5 : (d'=4)&(Dice'=10);
132     [] (Dice=8) → 0.5 : (d'=5)&(Dice'=10) + 0.5 : (d'=6)&(Dice'=10);
133     [] (Dice=10) → 1 : (Dice'=10);
134
135 endmodule
136

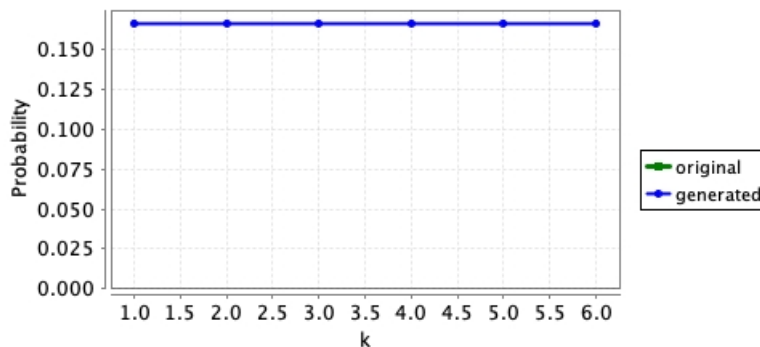
```

■ **Listing 2** Generated PRISM program for the Dice Program.

By comparing our model with the one presented in the PRISM documentation, we noticed that the difference is the number assumed by the variable `Dice`. In particular, the variable does not assume the values 1, 5 and 9. 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. Moreover, to prove the generated program is correct, we show that the probability of reaching a state where

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

137 The results are displayed in Figure 1, where also the results obtained with the original PRISM model are shown.



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

2.2 Simple Peer-To-Peer Protocol

This case study describes a simple peer-to-peer protocol based on BitTorrent². The model comprises a set of clients trying to download a file that has been partitioned into K blocks. Initially, there is one client that has already obtained all of the blocks and N additional clients with no blocks. Each client can download a block from any of the others but they can only attempt four concurrent downloads for each block. The code we analyze with $k = 5$ and $N = 4$ is reported in Listing 3.

```

146 preamble
147 "ctmc"
148 "const double mu=2;"
149 "formula rate1=mu*(1+min(3,b11+b21+b31+b41));"
150 "formula rate2=mu*(1+min(3,b12+b22+b32+b42));"
151 "formula rate3=mu*(1+min(3,b13+b23+b33+b43));"
152 "formula rate4=mu*(1+min(3,b14+b24+b34+b44));"
153 "formula rate5=mu*(1+min(3,b15+b25+b35+b45));"
154 endpreamble
155
156 n = 4;
157 n = 4;
158
159 Client[i] → i in [1..n]
160 Client[i] : "b[i]1 : [0..1];", "b[i]2 : [0..1];", "b[i]3 : [0..1];", "b[i]4 :
161           [0..1];", "b[i]5 : [0..1];" ;
162
163 {
164 PeerToPeer := Client[i] → Client[i]:
165           (+["rate1*1"] "(b[i]1'=1)"&&" " . PeerToPeer
166           +["rate2*1"] "(b[i]2'=1)"&&" " . PeerToPeer
167           +["rate3*1"] "(b[i]3'=1)"&&" " . PeerToPeer
168           +["rate4*1"] "(b[i]4'=1)"&&" " . PeerToPeer
169           +["rate5*1"] "(b[i]5'=1)"&&" " . PeerToPeer)
170 }
171 }
172

```

■ Listing 3 Choreographic language for the Peer-To-Peer Protocol.

Part of the generated PRISM code is shown in Listing 4 and it is faithful with what reported in the PRISM documentation.

```

173 ctmc
174 const double mu=2;
175 formula rate1=mu*(1+min(3,b11+b21+b31+b41));
176 formula rate2=mu*(1+min(3,b12+b22+b32+b42));
177 formula rate3=mu*(1+min(3,b13+b23+b33+b43));
178 formula rate4=mu*(1+min(3,b14+b24+b34+b44));
179 formula rate5=mu*(1+min(3,b15+b25+b35+b45));
180
181 module Client1
182   Client1 : [0..1] init 0;
183   b11 : [0..1];
184

```

² <https://www.prismmodelchecker.org/casestudies/peer2peer.php>

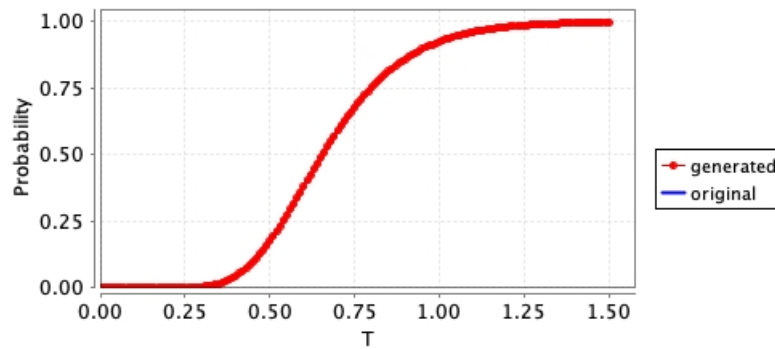

```

187      b12 : [0..1];
188      b13 : [0..1];
189      b14 : [0..1];
190      b15 : [0..1];
191
192      [] ( Client1=0 ) → rate1 : (b11'=1)&(Client1'=0);
193      [] ( Client1=0 ) → rate2 : (b12'=1)&(Client1'=0);
194      [] ( Client1=0 ) → rate3 : (b13'=1)&(Client1'=0);
195      [] ( Client1=0 ) → rate4 : (b14'=1)&(Client1'=0);
196      [] ( Client1=0 ) → rate5 : (b15'=1)&(Client1'=0);
197
198  endmodule
199

```

■ **Listing 4** Generated PRISM program for the Peer-To-Peer Protocol.

200 In Figure 2, we compare the values obtained for the probability that all clients have
 201 received all blocks by time $0 \leq T \leq 1.5$ both for our generated model and the model reported
 in the documentation.



■ **Figure 2** Probability that clients received all the block before T , with $0 \leq T \leq 1.5$.

202

203 2.3 Proof of Work Bitcoin Protocol

204 This protocol represents the Proof of Work implemented in the Bitcoin blockchain. In[1],
 205 a Bitcoin system is the result of the parallel composition of n Miner processes, n *Hasher*
 206 processes and a process called *Network*. *Hasher* processes model the attempts of the miners
 207 to solve the cryptopuzzle, while the *Network* process model the broadcast communication
 208 among miners. We tested our system by considering a protocol with $n = 5$ miners and it is
 209 reported in Listing 5.

```

210  preamble
211
212
213  "ctmc"
214
215  "const k;"
216  "const T=k*1000;"
217
218  "const double r = 1;"
219  "const double mR = 1/600;"

```

```

220 "const double lR = 1-mR;"
221
222 "const double hR1 = 0.25;"
223 "const double hR2 = 0.25;"
224 "const double hR3 = 0.25;"
225 "const double hR4 = 0.25;"
226 "const double rB = 1/12.6;"
227
228 "const int N = 100;"
229
230 endpreamble
231
232 n = 4;
233
234 Hasher[i] -> i in [1..n] ;
235
236 Miner[i] -> i in [1..n]
237 Miner[i] : "b[i] : block {m[i],0;genesis,0} ;", "B[i] : blockchain [{genesis,0;
238     genesis,0}];", "c[i] : [0..N] init 0;", "setMiner[i] : list [];" ;
239
240 Network ->
241 Network : "set1 : list [];", "set2 : list [];", "set3 : list [];" , "set4 : list
242     [];" ;
243
244 {
245 PoW := Hasher[i] → Miner[i] :
246 (+["mR*hR[i]"] " "&&"(b[i]’=createB(b[i],B[i],c[i]))&(c[i]’=c[i]+1)" .
247     Miner[i] → Network : ([rB*1] "(B[i]’=addBlock(B[i],b[i]))" &&
248     foreach(k != i) "(set[k]’=addBlockSet(set[k],b[i]))" @Network .
249     PoW)
250 +["lR*hR[i]"] " " && " " .
251     if "!isEmpty(set[i])"@Miner[i] then {
252         ["r"] "(b[i]’=extractBlock(set[i]))"
253         @Miner[i] . Miner[i] → Network :
254         ([1*1] "(setMiner[i]’ =
255         addBlockSet(setMiner[i] , b[i]))"
256         &&"(set[i]’ = removeBlock(set[i],b
257         [i]))" . PoW)
258     }
259     else{
260         if "canBeInserted(B[i],b[i])"@Miner[i] then {
261             ["1"] "(B[i]’=addBlock(B[i],b[i]))&(
262             setMiner[i]’=removeBlock(setMiner[i]
263             ,b[i]))"@Miner[i] . Pow
264         }
265         else{
266             PoW
267         }
268     }
269 }
270

```

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

271 Part of the generated PRISM code is shown in Listing 6.

272
273 **ctmc**

```

274  const k=1;
275  const T;
276  const double r = 1;
277  const double mR = 1/600;
278  const double lR = 1-mR;
279  const double hR1 = 0.25;
280  const double hR2 = 0.25;
281  const double hR3 = 0.25;
282  const double hR4 = 0.25;
283  const double rB = 1/12.6;
284  const int N = 100;
285
286  module Miner1
287  Miner1 : [0..7] init 0;
288  b1 : block {m1,0;genesis,0} ;
289  B1 : blockchain [{genesis,0;genesis,0}];
290  c1 : [0..N] init 0;
291  setMiner1 : list [];
292
293  [PZKYT] (Miner1=0) → hR1 : (b1'=createB(b1,B1,c1))&(c1'=c1+1)&(Miner1'=1);
294  [EUBVP] (Miner1=0) → hR1 : (Miner1'=2);
295  [HXYKO] (Miner1=1) → 1 : (B1'=addBlock(B1,b1))&(Miner1'=0);
296  [] (Miner1=2)&!isEmpty(set1) → r : (b1'=extractBlock(set1))&(Miner1'=4);
297  [SRKSV] (Miner1=4) → 1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Miner1'=0);
298  [] (Miner1=2)&!(!isEmpty(set1)) → 1 : (Miner1'=5);
299  [] (Miner1=5)&canBeInserted(B1,b1) → 1 : (B1'=addBlock(B1,b1))
300      &(setMiner1'=removeBlock(setMiner1,b1))&(Miner1'=0);
301  [] (Miner1=5)&!(!canBeInserted(B1,b1)) → 1 : (Miner1'=0);
302  endmodule
303
304
305  ...
306
307  module Network
308  Network : [0..1] init 0;
309  set1 : list [];
310  ...
311
312  [HXYKO] (Network=0) → 1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,b3))
313      &(set4'=addBlockSet(set4,b4))&(Network'=0);
314  [SRKSV] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
315  ...
316
317  endmodule
318
319
320  module Hasher1
321  Hasher1 : [0..1] init 0;

```

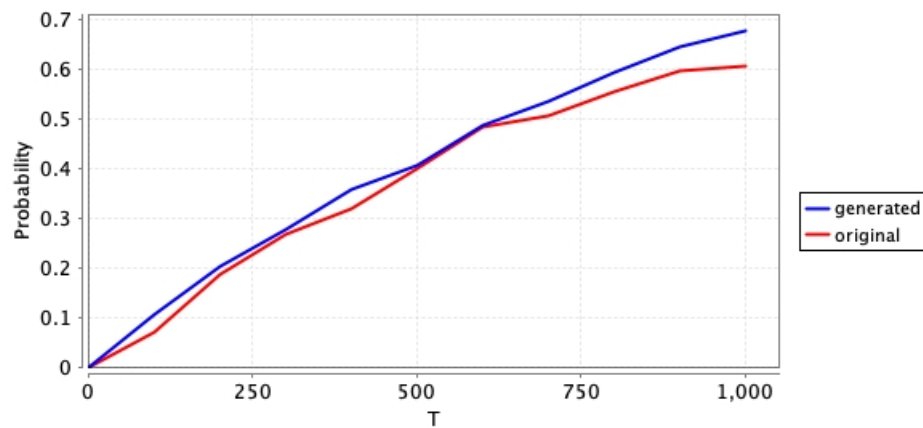
```

322
323 [PZKYT] (Hasher1=0) → mR : (Hasher1'=0);
324 [EUBVP] (Hasher1=0) → IR : (Hasher1'=0);
325
326 endmodule
327

```

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

328 In Figure 2, we compare the values obtained for the probability that at least one miner has mined a block both for the generated model and the model presented in [1].



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

329

330 References

- 331 1 Stefano Bistarelli, Rocco De Nicola, Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and
332 Adele Veschetti. Stochastic modeling and analysis of the bitcoin protocol in the presence of block
333 communication delays. *Concurr. Comput. Pract. Exp.*, 35(16), 2023. doi:10.1002/cpe.6749.