

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:10

Leibniz International Proceedings in Informatics



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

24 rules.

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

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

$$W(M) = \{F \mid F \in \{[M]\}\}$$

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

$$\sigma : X \rightarrow V$$

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.

$$f\left(\mathbf{p}_1 \longrightarrow \{\mathbf{p}_2\} \oplus \left\{ \begin{array}{l} [\lambda_1]x = 5 : \mathbf{p}_1 \longrightarrow \{\mathbf{p}_2\} \oplus \{[\lambda_3]y = 5\} \\ [\lambda_2]y = 10 : \mathbf{p}_1 \longrightarrow \{\mathbf{p}_2\} \oplus \{[\lambda_4]x = 10\} \end{array} \right\}, \mathbf{p}_1 : \emptyset \parallel \mathbf{p}_2 : \emptyset \right)$$

=

32 label = newlabel();
 for \mathbf{p}_i {
 $add(p_i, [label]s_{p_i} = state(p_i) \rightarrow \left\{ \begin{array}{l} \lambda_1 : x' = 5; state(\mathbf{p}_i)' = \text{generatenewstate}(\mathbf{p}_i) \\ \lambda_2 : y' = 10; state(\mathbf{p}_i)' = \text{generatenewstate}(\mathbf{p}_i) \end{array} \right\})$
 $f(\mathbf{p}_1 \longrightarrow \{\mathbf{p}_2\} \oplus \{[\lambda_3]y = 5\}, network') = network''$
 $return f(\mathbf{p}_1 \longrightarrow \{\mathbf{p}_2\} \oplus \{[\lambda_4]x = 10\}, network'')$

$$33 \quad f : C \longrightarrow \text{network} \longrightarrow \text{network} \quad \text{network} : \mathcal{R} \longrightarrow \text{Set}(F)$$

$$f\left(\mathbf{p}_1 \longrightarrow \{\mathbf{p}_i\}_{i \in I} \oplus \{[\lambda_j]x_j = E_j : D_j\}_{j \in J}, \mathbf{network}\right)$$

=

```

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

```

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

=

```

35 network = add( $\mathbf{p}$ , [ $s_{\mathbf{p}} = \text{state}(\mathbf{p}) \ \& \ f(E)$ ]);
network =  $f(C_1, \mathbf{network})$ ;
network =  $f(C_2, \mathbf{network})$ ;
return network

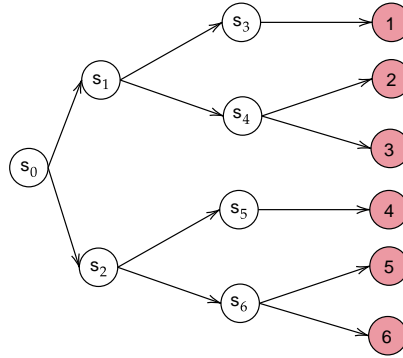
```

2 Tests

We tested our language by various examples.

2.1 The Dice Program

The first example we present is the Dice Program¹ [2]. 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.

```

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)

```

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

```

70
71 DiceProtocol3 := Dice → Dice : ([ "1*1" " "&" " ".DiceProtocol3)
72 }
73

```

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

```

74
75 dtmc
76
77 module Dice
78     Dice : [0..11] init 0;
79     d : [0..6] init 0;
80
81     [] (Dice=0) → 0.5 : (Dice'=2) + 0.5 : (Dice'=6);
82     [] (Dice=2) → 0.5 : (Dice'=3) + 0.5 : (Dice'=4);
83     [] (Dice=3) → 0.5 : (Dice'=2) + 0.5 : (d'=1)&(Dice'=10);
84     [] (Dice=4) → 0.5 : (d'=2)&(Dice'=10) + 0.5 : (d'=3)&(Dice'=10);
85     [] (Dice=6) → 0.5 : (Dice'=7) + 0.5 : (Dice'=8);
86     [] (Dice=7) → 0.5 : (Dice'=6) + 0.5 : (d'=4)&(Dice'=10);
87     [] (Dice=8) → 0.5 : (d'=5)&(Dice'=10) + 0.5 : (d'=6)&(Dice'=10);
88     [] (Dice=10) → 1 : (Dice'=10);
89
90 endmodule
91

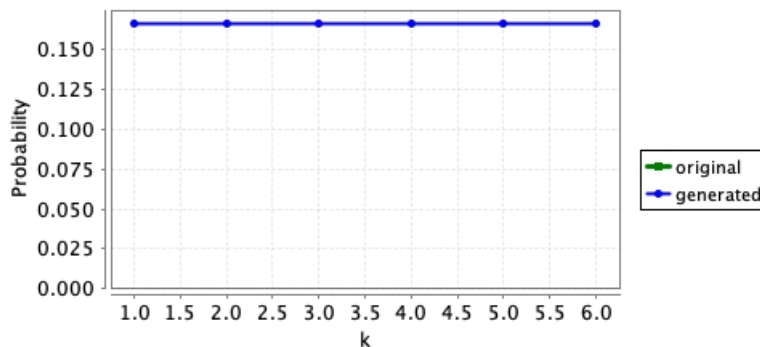
```

■ **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

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

92 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.

```

101 preamble
102 "ctmc"
103 "const double mu=2;"
104 "formula rate1=mu*(1+min(3,b11+b21+b31+b41));"
105 "formula rate2=mu*(1+min(3,b12+b22+b32+b42));"
106 "formula rate3=mu*(1+min(3,b13+b23+b33+b43));"
107 "formula rate4=mu*(1+min(3,b14+b24+b34+b44));"
108 "formula rate5=mu*(1+min(3,b15+b25+b35+b45));"
109 endpreamble
110
111
112 n = 4;
113 n = 4;
114
115 Client[i] → i in [1..n]
116 Client[i] : "b[i]1 : [0..1];", "b[i]2 : [0..1];", "b[i]3 : [0..1];", "b[i]4 :
117           [0..1];", "b[i]5 : [0..1];" ;
118
119 {
120 PeerToPeer := Client[i] → Client[i]:
121           (+["rate1*1"] "(b[i]1'=1)"&&" " . PeerToPeer
122           +["rate2*1"] "(b[i]2'=1)"&&" " . PeerToPeer
123           +["rate3*1"] "(b[i]3'=1)"&&" " . PeerToPeer
124           +["rate4*1"] "(b[i]4'=1)"&&" " . PeerToPeer
125           +["rate5*1"] "(b[i]5'=1)"&&" " . PeerToPeer)
126 }
127

```

■ **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.

```

130
131 ctmc
132 const double mu=2;
133 formula rate1=mu*(1+min(3,b11+b21+b31+b41));
134 formula rate2=mu*(1+min(3,b12+b22+b32+b42));
135 formula rate3=mu*(1+min(3,b13+b23+b33+b43));
136 formula rate4=mu*(1+min(3,b14+b24+b34+b44));
137 formula rate5=mu*(1+min(3,b15+b25+b35+b45));
138
139 module Client1
140   Client1 : [0..1] init 0;
141   b11 : [0..1];

```

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

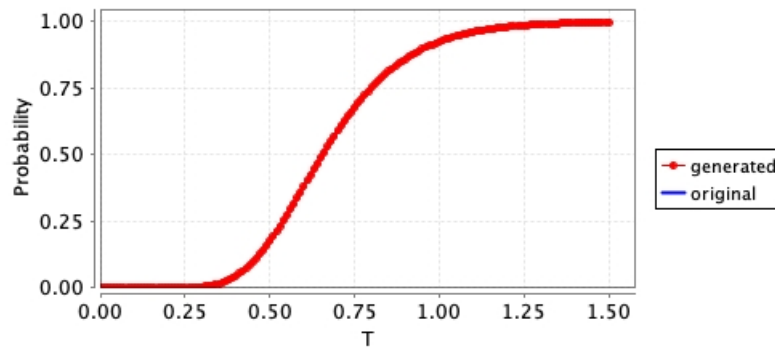
```

142      b12 : [0..1];
143      b13 : [0..1];
144      b14 : [0..1];
145      b15 : [0..1];
146
147      [] ( Client1=0 ) → rate1 : (b11'=1)&(Client1'=0);
148      [] ( Client1=0 ) → rate2 : (b12'=1)&(Client1'=0);
149      [] ( Client1=0 ) → rate3 : (b13'=1)&(Client1'=0);
150      [] ( Client1=0 ) → rate4 : (b14'=1)&(Client1'=0);
151      [] ( Client1=0 ) → rate5 : (b15'=1)&(Client1'=0);
152
153  endmodule
154

```

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

155 In Figure 2, we compare the values obtained for the probability that all clients have
 156 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 $0 \leq T \leq 1.5$.

157

158 2.3 Proof of Work Bitcoin Protocol

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

```

165  preamble
166  "ctmc"
167  "const T"
168  "const double r = 1;"
169  "const double mR = 1/600;"
170  "const double lR = 1-mR;"
171  "const double hR1 = 0.25;"
172  "const double hR2 = 0.25;"
173  "const double hR3 = 0.25;"
174

```

m:8 A Choreographic Language for PRISM

```

175 "const double hR4 = 0.25;"
176 "const double rB = 1/12.6;"
177 "const int N = 100;"
178 endpreamble
179
180 n = 4;
181
182 Hasher[i] -> i in [1..n] ;
183
184 Miner[i] -> i in [1..n]
185 Miner[i] : "b[i] : block {m[i],0;genesis,0} ;", "B[i] : blockchain [{genesis,0;
186     genesis,0}];", "c[i] : [0..N] init 0;", "setMiner[i] : list [];" ;
187
188 Network ->
189 Network : "set1 : list [];", "set2 : list [];", "set3 : list [];" , "set4 : list
190     [];" ;
191
192 {
193 PoW := Hasher[i] → Miner[i] :
194 (+["mR*hR[i]" " "&&"(b[i]'=createB(b[i],B[i],c[i]))&(c[i]'=c[i]+1)" " .
195     Miner[i] → Network :
196         ([ "rB*1" " "(B[i]'=addBlock(B[i],b[i]))" "&&
197         foreach(k != i) "(set[k]'=addBlockSet(set[k],b[i]))" @Network .PoW)
198 +["lR*hR[i]" " " "&&" " " .
199     if "!isEmpty(set[i])"@Miner[i] then {
200         ["r" " "(b[i]'=extractBlock(set[i]))"@Miner[i] .
201         Miner[i] → Network :
202         ([ "1*1" " "(setMiner[i]' = addBlockSet(setMiner[i] , b[i]))"
203         &&"(set[i]' = removeBlock(set[i],b[i]))" . PoW)
204     }
205     else{
206         if "canBeInserted(B[i],b[i])"@Miner[i] then {
207             ["1" " "(B[i]'=addBlock(B[i],b[i]))
208             &(setMiner[i]'=removeBlock(setMiner[i],b[i]))"@Miner[i] . Pow
209         }
210         else{
211             PoW
212         }
213     }
214 }
215 }
216

```

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

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

```

217
218
219 ctmc
220 const T;
221 const double r = 1;
222 const double mR = 1/600;
223 const double IR = 1-mR;
224 const double hR1 = 0.25;
225 const double hR2 = 0.25;
226 const double hR3 = 0.25;
227 const double hR4 = 0.25;

```



```

228 const double rB = 1/12.6;
229 const int N = 100;
230
231 module Miner1
232   Miner1 : [0..7] init 0;
233   b1 : block {m1,0;genesis,0} ;
234   B1 : blockchain [{genesis,0;genesis,0}];
235   c1 : [0..N] init 0;
236   setMiner1 : list [];
237
238   [PZKYT] (Miner1=0) → hR1 : (b1'=createB(b1,B1,c1))&(c1'=c1+1)&(Miner1'=1);
239   [EUBVP] (Miner1=0) → hR1 : (Miner1'=2);
240   [HXYKO] (Miner1=1) → 1 : (B1'=addBlock(B1,b1))&(Miner1'=0);
241   [] (Miner1=2)&!isEmpty(set1) → r : (b1'=extractBlock(set1))&(Miner1'=4);
242   [SRKSV] (Miner1=4) → 1 : (setMiner1' = addBlockSet(setMiner1 , b1))&(Miner1'=0);
243   [] (Miner1=2)&!isEmpty(set1) → 1 : (Miner1'=5);
244   [] (Miner1=5)&canBeInserted(B1,b1) → 1 : (B1'=addBlock(B1,b1))
245       &(setMiner1'=removeBlock(setMiner1,b1))&(Miner1'=0);
246   [] (Miner1=5)&!canBeInserted(B1,b1) → 1 : (Miner1'=0);
247 endmodule
248 ...
249 module Network
250   Network : [0..1] init 0;
251   set1 : list [];
252   ...
253
254   [HXYKO] (Network=0) → 1 : (set2'=addBlockSet(set2,b2))&(set3'=addBlockSet(set3,
255       b3))&(set4'=addBlockSet(set4,b4))&(Network'=0);
256   [SRKSV] (Network=0) → 1 : (set1' = removeBlock(set1,b1))&(Network'=0);
257   ...
258
259 endmodule
260
261 module Hasher1
262   Hasher1 : [0..1] init 0;
263
264   [PZKYT] (Hasher1=0) → mR : (Hasher1'=0);
265   [EUBVP] (Hasher1=0) → IR : (Hasher1'=0);
266
267 endmodule
268

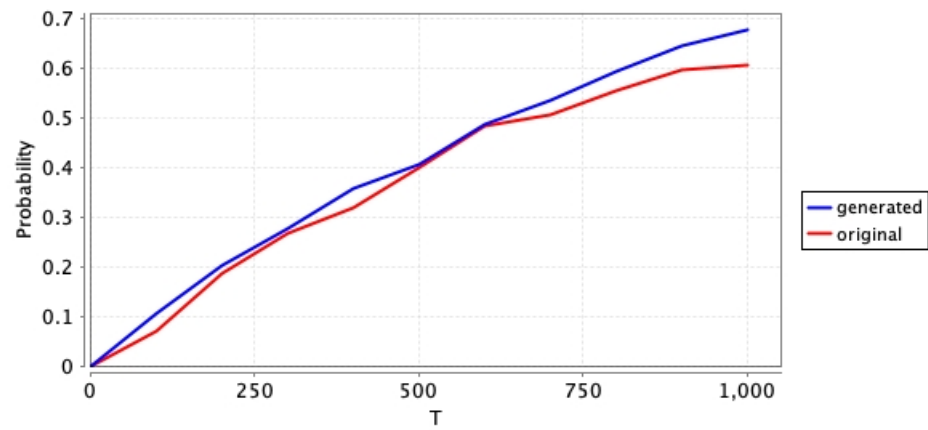
```

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

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

271 References

- 272 1 Stefano Bistarelli, Rocco De Nicola, Letterio Galletta, Cosimo Laneve, Ivan Mercanti, and
 273 Adele Veschetti. Stochastic modeling and analysis of the bitcoin protocol in the presence of block



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

- 274 communication delays. *Concurr. Comput. Pract. Exp.*, 35(16), 2023. doi:10.1002/cpe.6749.
- 275 2 D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter
- 276 The complexity of nonuniform random number generation. Academic Press, 1976.