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:

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

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

1.2 PRISM

Syntax.

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

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

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

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

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

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

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

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

$$\sigma : X \to V$$

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.

$$f : C \to \mathcal{R} \mapsto F$$

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

$$f : [C_1, \dots, C_n] \to String \mapsto String$$

```
CASE 1: C_i ≡ p₁ ⟶ {p_i}_{i∈I} ⊕ {[λ_j]x_j = E_j : C_j}_{j∈J}

   f([C_i,...,C_n], code) :
      label = generateNewLabel()
      for c_j in C_i :
         for p ∈ roles(C_i):
            newCode = "[label] (x_j = E_j)_p"
            code = code + newCode
      f([C_{i+1},...,C_n],code)
```

where $\mathbf{roles}(\mathrm{C}_i) \coloneqq \mathsf{p}_1 \cup \{\mathsf{p}_i\}_{i\in I}$.

```
CASE 2: C_i ≡ if E@p then C_1 else C_2

    f([C_i,...,C_n], code) :
        code = code + (E)_p
        f(C_1,code)
        f(C_2,code)
        f([C_{i+1},...,C_n],code)
```

$\mathtt{network} : \mathcal{R} \longrightarrow \mathtt{Set}(F)$

$\mathtt{f}(C_1, \ldots, C_n, \mathtt{network}) =$

```
CASE 1: ∀i.C_i  ≡  p_1 ⟶ {p_i}_{i∈I} ⊕ {[λ_j]x_j = E_j : D_j}_{j∈J}

 ->

    label = generateNewLabel()




   for c_j in C_i :
   for p ∈ roles(C_i):
   newCode = "[label] (x_j = E_j)_p"
   code = code + newCode
   f([C_{i+1},...,C_n],code)
```

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

$$=$$

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

$$f : C \longrightarrow \mathtt{network} \longrightarrow \mathtt{network} \qquad \mathtt{network} : \mathcal{R} \longrightarrow \mathrm{Set}(F)$$

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

$$=$$

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

$$f\Big(\text{ if } E@\mathsf{p} \text{ then } C_1 \text{ else } C_2, \texttt{network}\Big)$$

$$=$$

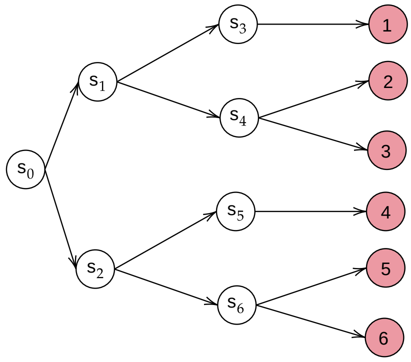
$$\begin{aligned}
&\texttt{network} = \texttt{add}(\mathsf{p}, [\,]s_{\mathsf{p}} = \texttt{state}(\mathsf{p})\ \&\ f(E));\\
&\texttt{network} = f(C_1, \texttt{network});\\
&\texttt{network} = f(C_2, \texttt{network});\\
&\mathrm{return}\ \texttt{network}
\end{aligned}$$

2 Tests

We tested our language by various examples.

2.1 The Dice Program

The first example we present is the Dice Program[1] [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

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

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 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

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

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

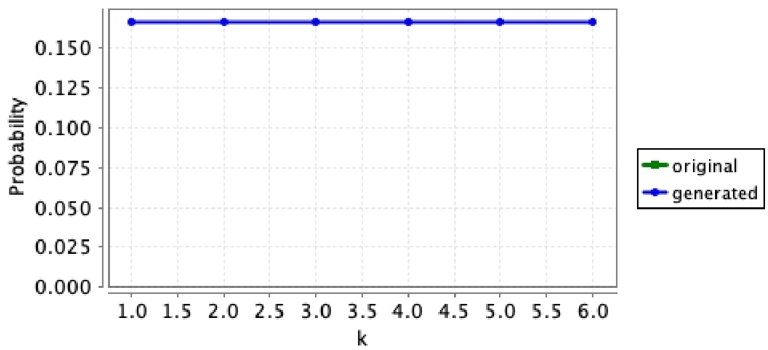


Figure 1 Probability of reaching a state where $d = k$, for $k = 1, \ldots, 6$.

2.2 Simple Peer-To-Peer Protocol

This case study describes a simple peer-to-peer protocol based on BitTorrent[2]. 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.

```
preamble
"ctmc"
"const double mu=2;"
"formula rate1=mu*(1+min(3,b11+b21+b31+b41));"
"formula rate2=mu*(1+min(3,b12+b22+b32+b42));"
"formula rate3=mu*(1+min(3,b13+b23+b33+b43));"
"formula rate4=mu*(1+min(3,b14+b24+b34+b44));"
"formula rate5=mu*(1+min(3,b15+b25+b35+b45));"
endpreamble

n = 4;
n = 4;

Client[i] → i in [1...n]
Client[i] : "b[i]1 : [0..1];", "b[i]2 : [0..1];", "b[i]3 : [0..1];", "b[i]4 :
    [0..1];", "b[i]5 : [0..1];" ;

{
PeerToPeer := Client[i] → Client[i]:
                    (+["rate1*1"] "(b[i]1'=1)"&&" " . PeerToPeer
                     +["rate2*1"] "(b[i]2'=1)"&&" " . PeerToPeer
                     +["rate3*1"] "(b[i]3'=1)"&&" " . PeerToPeer
                     +["rate4*1"] "(b[i]4'=1)"&&" " . PeerToPeer
                     +["rate5*1"] "(b[i]5'=1)"&&" " . PeerToPeer)
}
```

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.

```
ctmc
const double mu=2;
formula rate1=mu*(1+min(3,b11+b21+b31+b41));
formula rate2=mu*(1+min(3,b12+b22+b32+b42));
formula rate3=mu*(1+min(3,b13+b23+b33+b43));
formula rate4=mu*(1+min(3,b14+b24+b34+b44));
formula rate5=mu*(1+min(3,b15+b25+b35+b45));

module Client1
        Client1 : [0..1] init 0;
        b11 : [0..1];
```

[2] https://www.prismmodelchecker.org/casestudies/peer2peer.php

```
        b12 : [0..1];
        b13 : [0..1];
        b14 : [0..1];
        b15 : [0..1];

        [] (Client1=0) → rate1 : (b11'=1)&(Client1'=0);
        [] (Client1=0) → rate2 : (b12'=1)&(Client1'=0);
        [] (Client1=0) → rate3 : (b13'=1)&(Client1'=0);
        [] (Client1=0) → rate4 : (b14'=1)&(Client1'=0);
        [] (Client1=0) → rate5 : (b15'=1)&(Client1'=0);

endmodule
```

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

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

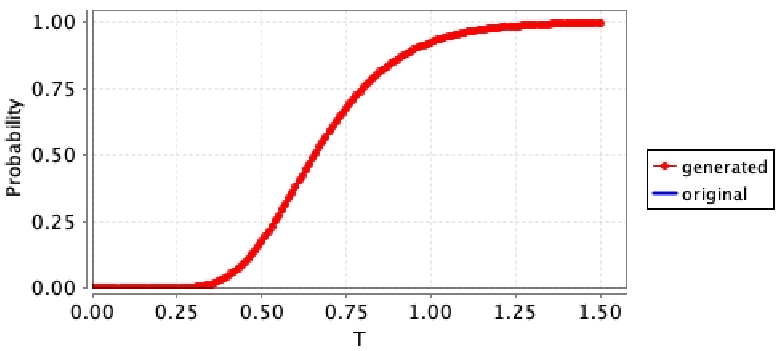


Figure 2 Probability that clients received all the block before T, with $0 \leq T \leq 1.5$.

2.3 Proof of Work Bitcoin Protocol

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

```
preamble
"ctmc"
"const T"
"const double r = 1;"
"const double mR = 1/600;"
"const double lR = 1-mR;"
"const double hR1 = 0.25;"
"const double hR2 = 0.25;"
"const double hR3 = 0.25;"
```

```
"const double hR4 = 0.25;"
"const double rB = 1/12.6;"
"const int N = 100;"
endpreamble

n = 4;

Hasher[i] -> i in [1...n] ;

Miner[i] -> i in [1...n]
Miner[i] : "b[i] : block {m[i],0;genesis,0} ;", "B[i] : blockchain [{genesis,0;
    genesis,0}];" ,"c[i] : [0..N] init 0;", "setMiner[i] : list [];" ;

Network ->
Network : "set1 : list [];", "set2 : list [];", "set3 : list [];" , "set4 : list
    [];";

{
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.

```
ctmc
const T;
const double r = 1;
const double mR = 1/600;
const double lR = 1−mR;
const double hR1 = 0.25;
const double hR2 = 0.25;
const double hR3 = 0.25;
const double hR4 = 0.25;
```

```
const double rB = 1/12.6;
const int N = 100;

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.

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].

References

1 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

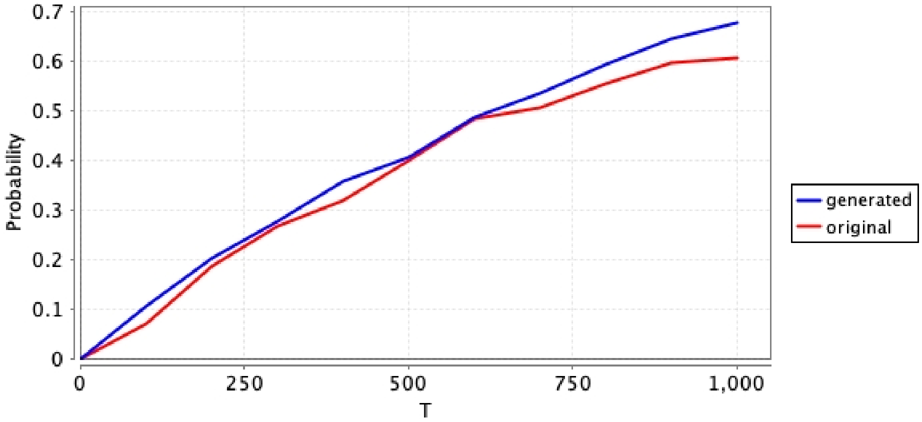


Figure 3 Probability at least one miner has created a block.

communication delays. *Concurr. Comput. Pract. Exp.*, 35(16), 2023. doi:10.1002/cpe.6749.

2 D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.