A Probabilistic Choreography Language for PRISM

Marco Carbone[1][0000−1111−2222−3333] and Adele Veschetti[2][1111−2222−3333−4444]

[1] IT University of Copenhagen
maca@itu.dk
[2] Technische Universität Darmstadt
adele.veschetti@tu-darmstadt.de

Abstract. The abstract should briefly summarize the contents of the paper in 150–250 words.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

This is the introduction

- introduce the PRISM language and use a running example.

- argue that these sorts of distributed programs are hard to understand and it is hard to programme them as intended since we may miss hidden cases.

- introduce the concept of choreography and show how the example in PRISM could be rewritten in a more concise way.

- End with a statement that says what this paper is doing.

Contributions and Overview. Our contributions can be categorised as follows:

–

2 The Prism Language

We start by describing the PRISM language syntax and 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:

(Networks)	$N, M ::=$	$\mathbf{0}$	empty network
		$\mid \mathsf{p} : \{F_i\}_i$	module
		$\mid M \vert [A] \vert M$	parallel composition
(Commands)	$F ::=$	$[a]g \to \Sigma_{i \in I}\{\lambda_i : u_i\}$	
(Assignment)	$u ::=$	$(x' = E)$	update x with E
		$\mid u \ \& \ u$	multiple assignments
(Expr)	$E, g ::=$	$f(\tilde{E}) \quad \mid \quad x \quad \mid \quad v$	expressions

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 $M_1 \vert [A] \vert M_2$ 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 have two possibilities: we can either proceed denotationally, following the approach given on the PRISM website [**?**] or define an operational semantics in the style of Plotkin [**?**] and Bookes et al. [**?**]. Since the semantics of the choreographic language we present is purely operational, we will opt for the second choice.

[HERE WE NEED FORMAL DEFINITIONS OF TRANSITION SYSTEM, Markov Chain, etc. But perhaps not because of space reason we can just claim that our semeantics is a Markov chain/process/whatever]

Definition 1 (Discrete Time Markov Chain (DTMC)). *A Discrete Time Markov Chain (DTMC) is a pair (S, P) where*

- *S is a set of states*
- *$P : S \times S \to [0, 1]$ is the probability transition matrix such that, for all $s \in S$, $\sum_{s' \in S} P(s, s') = 1$.*

Definition 2 (Continuous Time Markov Chain (CTMC)). *A Continuous Time Markov Chain (DTMC) is a pair* (S, R) *where*

- S *is a set of states*
- $P : S \times S \to R^{\geq 0}$ *is the rate transition matrix.*

in the sequel, we assume that in every command, each branch effect on any state is injective.

In the sequel, we assumet $\alpha \in \{\epsilon\} \cup \mathcal{L}$ where ϵ is the empty string. We now define the operational semantics as the minimum relation $\rightsquigarrow$ satisfying the following rules:

$$\frac{[]g \to \Sigma_{i\in I}\{\lambda_i : u_i\} \in \{F_k\}_k}{\mathsf{p} : \{F_k\}_k \;\rightsquigarrow\; [g][\lambda_i :\; u_i]}\ (\mathsf{M}_1) \qquad \frac{\exists j \in \{1,2\}.\ M_j \overset{\alpha}{\rightsquigarrow} [g][\lambda :\; u] \quad \alpha \notin A}{M_1|[A]|M_2 \overset{a}{\rightsquigarrow} [g][\lambda :\; u]}\ (\mathsf{Par}_1)$$

$$\frac{[a]g \to \Sigma_{i\in I}\{\lambda_i : u_i\} \in \{F_k\}_k}{\mathsf{p} : \{F_k\}_k \overset{a}{\rightsquigarrow} [g][\lambda_i :\; u_i]}\ (\mathsf{M}_2) \qquad \frac{M_1 \overset{a}{\rightsquigarrow} [g_1][\lambda_1 :\; u_1] \quad M_2 \overset{a}{\rightsquigarrow} [g_2][\lambda_2 :\; u_2] \quad a \in A}{M_1|[A]|M_2 \overset{a}{\rightsquigarrow} [g_1 \wedge g_2][\lambda_1 * \lambda_2 :\; u_1 \& u_2]}\ (\mathsf{Par}_2)$$

Then, we define the transition relation $M \vdash S \longrightarrow_\lambda S'$ as follows

$$\frac{\forall i, \alpha.\ M \overset{\alpha}{\rightsquigarrow} [g_i][\lambda_i :\; u_i] \qquad S \vdash g_i}{M \vdash S \longrightarrow_{\sum_{S'=S[u_j]} \lambda_j} S'}\ (\mathsf{Transition})$$

Note that if the λ's are probabilities, hence, we deal with an DTMC, then the probabilities of the branches outgoing from a give state S must be normalised – this is because the sum at the bottom of the Transition rule can be more than one. This is not the case if we have a CTMC with rates.

3 Choreographic Language

We now give syntax and semantics for our choreographic language. **Syntax.** Our choreographic language is defined by the following syntax:

$$\text{(Chor)} \quad C \;::=\; \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j\in J} \lambda_j : x_j = E_j;\ C_j \;\mid\; \mathsf{if}\ E@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \;\mid\; X \;\mid\; \mathbf{0}$$

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]

Semantics. Similarly to how we did for the PRISM language, we consider the state space Val^n where n is the number of variables present in the choreography. We then inductively define the transition function for the state space as follows:

$$(S, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j\in J} \lambda_j : x_j = E_j;\ C_j) \;\longrightarrow_{\lambda_j}\; (S[\sigma(E_j)/x_j], C_j)$$

$$(S, \mathsf{if}\ E@\mathsf{p}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) \;\longrightarrow\; (S, C_1)$$

$$X \overset{\mathsf{def}}{=} C \quad \Rightarrow \quad (S, X) \;\longrightarrow\; (S, C)$$

From the transition relation above, we can immediately define an LTS on the state space. Given an initial state σ_0 and a choreography C, the LTS is given by all the states reachable from the pair (σ_0, C). I.e., for all derivations $(\sigma_0, C) \longrightarrow_{\lambda_0} \ldots \longrightarrow_{\lambda_n} (\sigma_n, C_n)$ and $i < n$, we have that $(\sigma_i, \sigma_{i+1}) \in \delta$ [adjust once the definition of probabilistic LTS is in].

3.1 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.

$$
\begin{array}{l}
\big(q \in \{\mathsf{p}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}, J = \{1, \ldots, n\},\ l_1, \ldots, l_n \text{ fresh}\big) \\
\mathsf{proj}(q, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j \in J} \lambda_j : x_j = E_j;\ C_j, s) = \\
\quad \Big\{ [l_j]\ s_q = s \to \kappa_j :\ s_q = s_q + 1 + \textstyle\sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ E_j \downarrow_q \Big\}_{j \in J} \quad \cup \\
\quad \bigcup_j \mathsf{proj}(q, C_j, s + 1 + \textstyle\sum_{k=1}^{j-1} \mathsf{nodes}(C_k))
\end{array}
$$

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

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

3.2 Correctness

In the sequel, S_+ is a state S extended with extra variables used by the projection.

Theorem 1 (EPP). *Given a well-formed choreography C, we have that $(S, C) \longrightarrow_\lambda (S', C')$ iff $\mathsf{proj}(*, C) \vdash S \uplus S_+ \longrightarrow_\lambda S' \uplus S'_+$.*

Proof. We prove each direction separately.

– (only if). Assume that

$$(S, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j \in J} \lambda_j : x_j = E_j;\ C_j) \longrightarrow_{\lambda_j} (S[\sigma(E_j)/x_j], C_j)$$

and let us consider the projection of the term

$$\mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j \in J} \lambda_j : x_j = E_j;\ C_j$$

Given some fresh $l_1, \ldots, l_n$, we generate the following commands for each q in $\{\mathsf{p}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}$:

$$\Big\{ [l_j]\ s_q = s \to \kappa_j :\ s_q = s_q + 1 + \sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ E_j \downarrow_q \Big\}_{j \in J}$$

where $\kappa_j = \lambda_j$ if $q \neq \mathsf{p}$ and $\kappa_j = 1$ otherwise. Because the labels l_j are fresh and the state counter is unique to this interaction, these are the only commands that can synchronise together: this can be shown from the rules defining the semantics of PRISM. Additionally, since all rates are set to 1 besides the commands generated for role p, the transition will also be labelled with λ_j.

– (if). In the opposite direction, we have that

$$\mathsf{proj}(*, \mathsf{p} \to \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}\, \Sigma_{j \in J} \lambda_j : x_j = E_j;\ C_j) \vdash S \uplus S_+ \ \longrightarrow_\lambda \ S' \uplus S'_+$$

Again, given some fresh $l_1, \ldots, l_n$, the projection that reduces must be such that for each q in $\{\mathsf{p}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}$:

$$\left\{ [l_j]\ s_q = s \to \kappa_j :\ s_q = s_q + 1 + \sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ E_j \downarrow_q \right\}_{j \in J}$$

where $\kappa_j = \lambda_j$ if $q \neq \mathsf{p}$ and $\kappa_j = 1$ otherwise. Again, the freshness of the labels l_j together with the working states $s_q q$
Because the labels l_j are fresh and the state counter is unique to this interaction, these are the only commands that can synchronise together: this can be shown from the rules defining the semantics of PRISM. Additionally, since all rates are set to 1 besides the commands generated for role p, the transition will also be labelled with λ_j.

4 Benchmarking

In this section we present the implementation and 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[3]; the Bitcoin proof of work protocol and the Hybrid Casper protocol, presented in [3, 5].

4.1 Implementation

We implemented our language in 1246 lines of Java. We defined the grammar of our language and generated both the parser and the visitor components with ANTLR [1]. Each abstract syntax tree (AST) node, such as `ActionNode`, `IfThenElseNode`, `BranchNode`, `InternalActionNode`, `LoopNode`, `MessageNode`, `ModuleNode`, `PreambleNode`, `ProtocolNode`, `RecNode`, `RoleNode`, and `ProgramNode`, was encapsulated in a corresponding `Node` class. The methods of these classes were then utilized to generate PRISM code.

```
String generateCode(ArrayList<Node> mods, int index, int maxIndex, boolean isCtmc, ArrayList<String>
    ↪ labels, String prot);
```

Listing 1.1. The `generateCode` function.

The `generateCode` function generates the projection from our language to PRISM. The input parameters for the projection function include:

- `mods`: a list containing the choreography modules. As new commands are generated, they are appended to the set of commands for the respective module;
- `index` and `maxIndex`: indices used to keep track of the currently analyzed role;
- `isCtmc`: a boolean flag indicating whether a Continuous-Time Markov Chain (CTMC) is being generated. This flag is crucial as the projection generation logic depends on it;
- `labels`: the pre-existing labels. This parameter is necessary for checking the uniqueness of a label;
- `prot`: the name of the protocol currently under analysis.

The projection function operates recursively on each command in the choreographic language, systematically generating PRISM code based on the type of command being analyzed. While most code generations are straightforward, the focal point lies in how new states are created. Each module maintains its set of states, and when a new state is to be generated, the function examines the last available state for the corresponding module and increments it by one. Recursion follows a similar pattern: every module possesses a table that accumulates recursion protocols, along with the first and last states associated with each recursion. This recursive approach ensures a systematic and coherent generation of states within the modules, enhancing the overall efficiency and clarity of the projection function.

[3] `https://www.prismmodelchecker.org/casestudies/`

4.2 The Dice Program

The first test case we focus on the Dice Program[4] [6]. The following program simulates a die using only unbiased coins. Commencing from the initial vertex (state s_0), the process involves the iterative flipping of a coin. Upon obtaining heads, the upper branch is chosen, and in the case of tails, the lower branch is taken. This sequential process persists until the final determination of the die's value. In Listing 1.2, we provide part of the program utilizing the choreographic language.

```
...
{
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)"&&" " . DiceProtocol7)
              +["0.5*1"] " "&&" " . Dice → Dice :
                     (+["0.5*1"] "(d'=2)"&&" " . DiceProtocol7
                     +["0.5*1"] "(d'=3)"&&" " . DiceProtocol7))
...
DiceProtocol7 :=
       Dice → Dice : (["1*1"] " "&&" ". END)
}
```

Listing 1.2. Choreographic language for the Dice Program.

By comparing the generated model with the one presented in the PRISM documentation, we notice that the difference is the number assumed by the variable `Dice`. Specifically, this variable takes on different values due to the manner in which generation occurs in the presence of a branch. Fortunately, this discrepancy doesn't lead to any issues, as the updates are executed accurately, and the states remain distinct.

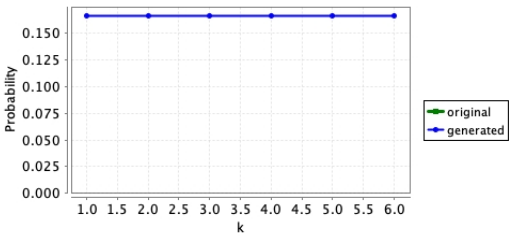


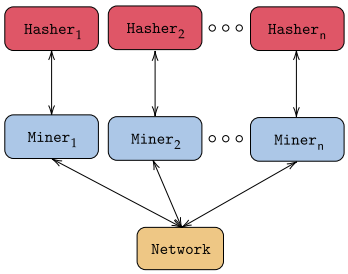
Fig. 1. Probability of reaching a state where $d = k$, for $k = 1, \ldots, 6$.

Furthermore, to establish the correctness of the generated program, we test that the probability of reaching a state where `d=k` for `k` $= 1, \ldots, 6$ is $\frac{1}{6}$. The outcomes are depicted in Figure 1, where we compare the probabilities obtained from our generated model with those from the original PRISM model. As expected, the results align perfectly.

4.3 Proof of Work Bitcoin Protocol

[4] `https://www.prismmodelchecker.org/casestudies/dice.php`

In the work presented in [3], the authors chose to enhance the capabilities of the PRISM model checker by incorporating dynamic data types. This extension aimed to effectively model the Proof of Work protocol implemented in the Bitcoin blockchain [7].



The Bitcoin system is modeled as a parallel composition of n Miner processes, n *Hasher* processes, and a process denoted as *Network*. Specifically:

- The *Miner* processes replicate blockchain miners responsible for generating new blocks and appending them to their local ledger.
- The *Hasher* processes simulate the process of solving the cryptographic puzzle.
- The *Network* process models the broadcast communication among miners.

As our focus lies not in the properties derived from analyzing the protocol, we opted to consider $n = 4$ miner and hasher processes.

```
...
{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 1.3. Choreographic language for the Proof of Work Bitcoin Protocol.

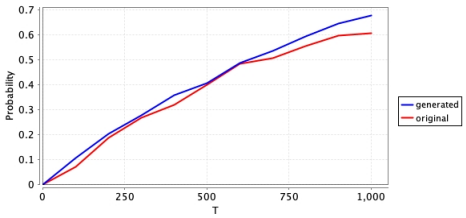
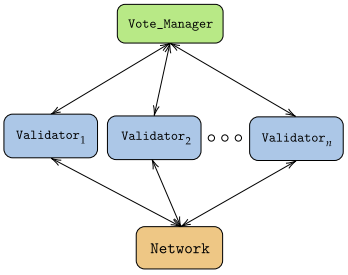


Fig. 2. Probability at least one miner has created a block.

The PRISM model we generated exhibits greater verbosity compared to the one presented in [3]. This difference arises from our approach of consistently generating the `else` branch for the `if-then-else` expression, resulting in an increased number of instructions. Nevertheless, in the case of this specific test scenario, the experimental results remain unaffected, as illustrated in Figure 2, where a comparison of the outcomes is presented. In this instance, the slight disparity in probabilities between the two results is unrelated to the model itself, but it is due to the fact that we are comparing two simulations.

4.4 Hybrid Casper Protocol



We now present the Hybrid Casper Protocol as modeled in PRISM, as detailed in [5]. The Hybrid Casper protocol represents a hybrid consensus protocol for blockchains, merging features from both Proof of Work and Proof of Stake protocols. Initially, it was implemented in the Ethereum blockchain [4] as a testing phase before switching to a pure Proof of Stake protocol.

The modeling approach is very similar to the one used for the Proof of Work Bitcoin protocol. Specifically, the Hybrid Casper protocol is represented in PRISM as the parallel composition of n `Validator` modules, along with the modules `Vote_Manager` and `Network`. The `Validator` module closely resembles the `Miner` module from the preceding protocol. The only module necessitating clarification is `Vote_Manager`, responsible for storing tables containing votes for each checkpoint and computing associated rewards/penalties. The modeling language is reported in Listing 1.4.

AV: Riscrivere/sistemare codice

```
...
{
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] -> Vote_Manager :(["1*1"] " "&&"(Votes'=addVote(Votes,b[i],stake[i]))".PoS
            ↪ )
    }
 +["hR*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{
                                 Validator[i] -> Network : (["1*1"] "(setMiner[i]' =
                                       ↪ addBlockSet(setMiner[i] , b[i]))"&&"(set[i]' =
                                       ↪ removeBlock(set[i],b[i]))" .
                                 if "!(mod(getHeight(b[i]),EpochSize)=0)"@Validator[i] then {
```

```
                                    PoS
                                }
                                else{
                                    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 1.4. Choreographic language for the Hybrid Casper Protocol.

The code closely resembles the one outlined in [5], with the main distinction being the greater number of lines in our generated model. This difference is due to the fact that certain commands that could be combined, but our generation lacks the automatic capability to perform this check. The divergence between the two models is evident in simulations, as depicted in Figure 3. While the results exhibit similarity, running simulations for the generated model takes PRISM 39.016 seconds, compared to the 22.051 seconds required for the original model.

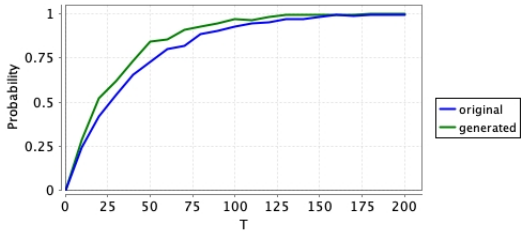


Fig. 3. Probability that a block has been created.

4.5 Problems

While testing our choreographic language, we noticed that some of the case studies presented in the PRISM documentation [2] 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 `station`$_i$ do two different things in the same state. More precicely, at the state 0 (`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 Discussion

Acknowledgments. The work is partially funded by H2020-MSCA-RISE Project 778233 (BEHAPI) and by the ATHENE project "Model-centric Deductive Verification of Smart Contracts".

References

1. ANTLR - another tool for language recognition. `https://www.antlr.org/`
2. Prism documentation. `https://www.prismmodelchecker.org/`, accessed: 2023-09-05
3. Bistarelli, S., Nicola, R.D., Galletta, L., Laneve, C., Mercanti, I., Veschetti, A.: Stochastic modeling and analysis of the bitcoin protocol in the presence of block communication delays. Concurr. Comput. Pract. Exp. **35**(16) (2023). https://doi.org/10.1002/cpe.6749, `https://doi.org/10.1002/cpe.6749`
4. Buterin, V.: Ethereum white paper. `https://github.com/ethereum/wiki/wiki/White-Paper` (2013)
5. Galletta, L., Laneve, C., Mercanti, I., Veschetti, A.: Resilience of hybrid casper under varying values of parameters. Distributed Ledger Technol. Res. Pract. **2**(1), 5:1–5:25 (2023). https://doi.org/10.1145/3571587, `https://doi.org/10.1145/3571587`

[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`

6. Knuth, D., Yao, A.: Algorithms and Complexity: New Directions and Recent Results, chap. The complexity of nonuniform random number generation. Academic Press (1976)
7. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf` (2008)

A A denotational semantics for PRISM

We proceed by steps. First, we define $\{\![-]\!\}$, as the closure of the following rules:

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

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

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

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

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

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

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

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.