A Probabilistic Choreography Language for PRISM

Marco Carbone$^{1[0000-0001-9479-2632]}$ and Adele Veschetti$^{2[0000-0002-0403-1889]}$

^{1} IT University of Copenhagen
`maca@itu.dk`
^{2} Technische Universität Darmstadt
`adele.veschetti@tu-darmstadt.de`

Abstract. We present a choreographic framework for modelling and analysing concurrent probabilistic systems based on the PRISM model-checker. This is achieved through the development of a choreography language, which is a specification language that allows to describe the desired interactions within a concurrent system from a global viewpoint. Employing choreographies provides a clear and comprehensive view of system interactions, enabling the discernment of process flow and detection of potential errors, thus ensuring accurate execution and enhancing system reliability. We equip our language with a probabilistic semantics and then define a formal encoding into the PRISM language and discuss its correctness. Properties of programs written in our choreographic language can be model-checked by the PRISM model-checker via their translation into the PRISM language. Finally, we implement a compiler for our language and demonstrate its practical applicability via examples drawn from the use cases featured in the PRISM website.

1 Introduction

Understanding and programming distributed systems pose formidable challenges due to their inherent complexity and the potential for elusive edge cases within their intricate interactions. Unlike monolithic systems, distributed programs involve multiple nodes operating concurrently and communicating over networks, introducing a multitude of potential failure scenarios and nondeterministic behaviours. One of the primary challenges in understanding distributed systems lies in the fact that the interactions between multiple components can diverge from the sum of their individual behaviours. This emergent behaviour often results from subtle interactions between nodes, making it difficult to predict and reason about the system's overall behaviour.

PRISM [2] is a probabilistic model checker that offers a specialised language for the specification and verification of probabilistic concurrent systems. PRISM has been used in various fields, including multimedia protocols [19], randomised distributed algorithms [16, 18], security protocols [22, 17], and biological systems [10, 13]. At its core, PRISM provides a declarative language with a set of constructs for describing probabilistic behaviours and properties within a system.

Given a distributed system, we can use PRISM to model the behaviour of each of its nodes, and then verify desired properties for the entire system. However, this approach can become difficult to manage as the number of nodes increases.

Choreographic programming [20] is an emerging programming paradigm in which programs, referred to as choreographies, serve as specifications providing a global perspective on the communication patterns inherent in a distributed system. In particular, instead of relying on a central orchestrator or controller to dictate the behavior of individual components, choreographic languages focus on defining communication patterns and protocols that govern the interactions between entities. In essence, choreographies abstract away the internal details of individual components and emphasise the global behaviour as a composition of decentralised interactions. This approach facilitates the automatic generation of decentralised implementations that are inherently correct-by-construction.

This paper presents a choreographic language designed for modelling concurrent probabilistic systems. Additionally, we introduce a compiler capable of translating protocols described in this language into PRISM code. This choreographic approach not only simplifies the modelling process but also ensures integration with PRISM's powerful analysis capabilities.

As an example, consider the following simplified version of a system presented in the PRISM documentation[3], where a user moves between different states (0, 1, or 2) based on certain events (α, β, γ) with corresponding rates (λ, μ, θ), and where a checkout process transitions between two states (0 or 1).

```
1        ctmc
2        module User
3                User_STATE : [0..2]  init  0;
4
5                [alpha_1] (User_STATE=0) → lambda : (User_STATE'=1);
6                [alpha_2] (User_STATE=0) → lambda : (User_STATE'=2);
7                [beta]  (User_STATE=1) → mu : (User_STATE'=0);
8                [gamma_1] (User_STATE=2) → theta : (User_STATE'=1);
9                [gamma_2] (User_STATE=2) → theta : (User_STATE'=2);
10       endmodule
11
12       module CheckOut
13               CheckOut_STATE : [0..1] init 0;
14
15               [alpha_1,alpha_2] (CheckOut_STATE=0) → 1 : (CheckOut_STATE'=1);
16               [beta]  (CheckOut_STATE=1) → 1 : (CheckOut_STATE'=0);
17               [gamma_1,gamma_2] (CheckOut_STATE=1) → 1 : (CheckOut_STATE'=1);
18       endmodule
```

In PRISM, modules are individual processes whose behaviour is specified by a collection of commands, in a declarative fashion. Processes have a local state, can interact with other modules and query each other's state. Above, the modules User (lines 2-10) and CheckOut (lines 12-18) synchronise on labels alpha_1, alpha_2, beta, gamma_1 and gamma_2. On line 5, (User_STATE=0) is a condition indicating that this transition is enabled when User_STATE has value 0. The variable lambda represents a rate, since the program models a Continuous Time Markov Chain (CTMC). The command (User_STATE'=1) is an update, indicating that User_STATE changes to 1 when this transition fires.

[3] https://www.prismmodelchecker.org/casestudies/thinkteam.php

Understanding the interactions between processes in this example might indeed be challenging, especially without additional context or explanation. Alternatively, when formalised using our choreographic language, the same model becomes significantly clearer.

```
C0 := User → Check : (+["lambda*1"] ; C1 +["lambda*1"] ; C2)
C1 := User → Check : (+["beta*1"] ; C0)
C2 := User → Check : (+["mu*1"] ; C1 +["mu*1"] ; C2)
```

In this model, we define three distinct choreographies, namely `C0`, `C1`, and `C2`. These choreographies describe the interaction patterns between the modules `User` and `Check`. The state updates resulting from these interactions are not explicitly depicted as they are not relevant for this particular protocol, but necessary in the PRISM implementation. As evident from this example, the choreographic language facilitates a straightforward understanding of the interactions between processes, minimizing the likelihood of errors.

Through our contributions, we aim to provide a smooth workflow for modeling, analyzing, and verifying concurrent probabilistic systems, ultimately increasing their usability in various application domains. In particular, by employing choreographies, we gain a clear and comprehensive view of the interactions occurring within the system, allowing us to discern the flow of processes and detect any potential sources of error in the modelling phase.

Contributions and Overview. Our contributions can be categorised as:

– Firstly, we propose a choreographic language equipped with well-defined syntax and semantics, tailored specifically for describing concurrent systems with probabilistic behaviours (§ 2). To the best of our knowledge, this is the first probabilistic choreography language that is not a type abstraction.
– Then, we introduce a semantics for PRISM (§ 3), based on the original semantics for PRISM [2].
– Furthermore, we establish a rigorous definition for a translation function from choreographies to PRISM (§ 4), and address its correctness. This translation (projection) serves as a crucial intermediary step in transforming (compiling) models described in our choreographic language into PRISM-compatible representations.
– Lastly, we implement a compiler that translates choreographies into PRISM, allowing users to use PRISM's robust analysis features while benefiting from the expressiveness of our choreographic language.

2 Choreography Language

In the introduction, we showed our choreography language via an example. We now formalise our language by giving its syntax and semantics. For the sake of clarity, we slightly change the syntax with respect to the syntax of our tool.

Syntax. Let $\mathtt{p}$ range over a (possibly infinite) set of module names $\mathcal{R}$, x over a (possibly infinite) set of variables Var, and v over a (possibly infinite) set of values Val. Choreographies, the key component of our language, are defined by the following syntax:

$$
\begin{array}{lll}
C & ::= \; \mathtt{p} \to \{\mathtt{p_1}, \dots, \mathtt{p_n}\} : \Sigma_{j\in J} \lambda_j : u_j;\ C_j \;\mid\; \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \;\mid\; X \;\mid\; \mathbf{0} & \\
u & ::= (x' = E)\ \&\ u \quad\mid\quad (x' = E) \qquad\qquad E, g ::= f(\tilde{E}) \quad\mid\quad x \quad\mid\quad v &
\end{array}
$$

The syntactic category C denotes choreographic programs. The interaction term $\mathtt{p} \to \{\mathtt{p}_1, \dots, \mathtt{p}_n\} : \Sigma\{\lambda_j : x_j = E_j.\ C_j\}_{j\in J}$ denotes an interaction initiated by module $\mathtt{p}$ with modules $\mathtt{p}_i$'s. A choreography specifies what interaction must be executed next, shifting the focus from what can happen to what must happen. When the interaction happens, one of the i branches is selected as a continuation. Branching is a random move: the number $\lambda_i \in \mathbb{R}$ denotes either a probability or a rate. This will depend on the language we wish to use. In the case of probabilities, it must be the case that $0 \leq \lambda_i \leq 1$ and $\Sigma_i \lambda_i = 1$. Once a branch i is taken, the choreography will execute some assignments u_i. A single assignment has the syntax $(x' = E)$ meaning that the value obtained by evaluating expression E is assigned to variable x. Note that x' is used for an assignment to x: here, we follow the syntax adopted in PRISM (see § 3). Expressions are obtained by applying some unspecified functions to other expressions. Their base terms are variables and values (denoted by v).

The term $\mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2$ denotes a system where module $\mathtt{p}$ evaluates the guard E (which can contain variables located at other modules) and then (deterministically) branches accordingly. The term X is a (possibly recursive) procedure call: in the semantics, we assume that such procedure names are defined separately. The term $\mathbf{0}$ denotes the system finishing its computation.

Semantics. In the sequel, we define a state, denoted by S, as a mapping from variables to values, i.e., $S : \mathsf{Var} \to \mathsf{Val}$. Given a value v and a variable x, a substitution $[v/x]$ is an update on a state, i.e., $S[v/x](y) = \begin{cases} v & \text{if } y = x \\ S(y) & \text{otherwise} \end{cases}$ Then, the update $S[u]$ is such that $S[x' = E\ \&\ u] = S[E{\downarrow_S}\ /x][u]$ and $S[x' = E] = S[E{\downarrow_S}\ /x]$, where $E{\downarrow_S}$ is an unspecified (decidable) evaluation of the expression E in the state S.

Given the set of all possible states $\mathcal{S}$ and a set of definitions Σ of the form $X \overset{\mathsf{def}}{=} C$, we can define the operational semantics of choreographies as the minimal relation $\longrightarrow^{\Sigma} \subseteq \mathcal{S} \times C \times \mathbb{R} \times \mathcal{S} \times C$ such that (we omit Σ):

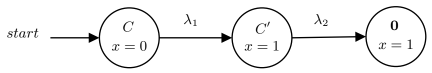
$$
\begin{array}{ll}
\textsf{(Interact)} & (S, \mathtt{p} \to \{\mathtt{p_1}, \dots, \mathtt{p_n}\} : \Sigma_{j\in J} \lambda_j : u_j;\ C_j) \ \longrightarrow_{\lambda_j}\ (S[u_j], C_j) \\
\textsf{(IfThenElseT)} & E{\downarrow_S} = \mathsf{tt} \quad\Rightarrow\quad (S, \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) \ \longrightarrow_1\ (S, C_1) \\
\textsf{(IfThenElseF)} & E{\downarrow_S} = \mathsf{ff} \quad\Rightarrow\quad (S, \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) \ \longrightarrow_1\ (S, C_2) \\
\textsf{(Call)} & X \overset{\mathsf{def}}{=} C \in \Sigma \quad\Rightarrow\quad (S, X) \ \longrightarrow_1\ (S, C)
\end{array}
$$

The transition relation is a Discrete Time Markov Chain (DTMC) or a Continuous Time Markov Chain (CTMC) depending on whether we use probabilities or rates in the branching construct. Note that states of the Markov chain are the pairs (S, C), while the transitions are given by the relation $\longrightarrow$.

Example 1. Consider the following choreography:

$$C = \mathtt{p} \to \{\mathtt{q}\}\ \lambda_1 : (x' = 1);\quad \mathtt{p} \to \{\mathtt{q}\}\ \lambda_2 : (x' = 1); \mathbf{0}$$

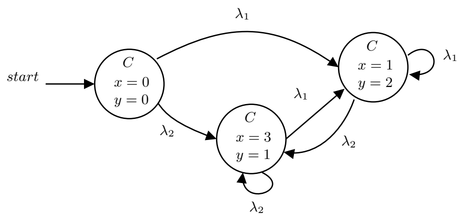
The semantics of C starting from a state in which $S(x) = S(y) = 0$ can be depicted as follows (for $C' = \mathtt{p} \to \{\mathtt{q}\}\ \lambda_1 : (x' = 1); \mathbf{0}$):



Example 2. Consider the following definition:

$$C \overset{\mathsf{def}}{=} \mathtt{p} \to \{\mathtt{q}\} \begin{cases} \lambda_1 : (x' = 1)\&(y' = 2);\ C \\ \lambda_2 : (x' = 3)\&(y' = 1);\ C \end{cases}$$

The semantics of C starting from a state in which $S(x) = S(y) = 0$ can be depicted as follows:



3 The PRISM Language

We now give a formal definition of a fragment of the PRISM language by introducing its formal syntax and semantics.

Syntax. We reuse some of the syntactic terms used for our choreography language, including assignments and expressions. In the sequel, let a range over a (possibly infinite) set of labels $\mathcal{L}$. We define the syntax of (a subset of) the PRISM language as follows:

$$\begin{array}{llll}
\text{(Networks)} & N, M ::= & \mathbf{0} & \text{empty network} \\
 & & \mid \mathtt{p} : \{F_i\}_i & \text{module} \\
 & & \mid M|[A]|M & \text{parallel composition} \\
 & & & \\
\text{(Commands)} & F ::= & [\alpha]\ g\ \to \Sigma_{i \in I} \lambda_i : u_i & (\alpha \in \{\epsilon\} \cup \mathcal{L})
\end{array}$$

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 name and its commands, formally written as $\mathtt{p} : \{F_i\}_i$, where $\mathtt{p}$ is the name and the F_i's are commands. 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 synchronise using labels in the finite set A. In this work, we omit PRISM's hiding and substitution constructs as they are irrelevant for our current choreography language. Commands in a module have the form $[\alpha]g \to \Sigma_{i\in I}\{\lambda_i : u_i\}$. The character α can either be the empty string ϵ or a label a, i.e., $\alpha \in \{\epsilon\} \cup \mathcal{L}$. If ϵ then no synchronisation is required. On the other hand, if there is label a then there will be a synchronisation with other modules that must synchronise on a. The term g is a guard on the current variable state. If both label and guard are enabled, then the command executes a branch i with probability/rate λ_i. As for choreographies, if the λ_i's are probabilities, we must have that $0 \leq \lambda_i \leq 1$ and $\Sigma_{i\in I}\lambda_i = 1$.

Semantics. To give a probabilistic semantics to the PRISM language, we follow the approach given in the PRISM documentation [2]. Hereby, we do that by defining two relations: one with labels for networks and one on states. Our relation on networks is the minimum relation $\rightsquigarrow$ satisfying the rules given in Fig. 1. Rule (M) just exposes a command at network level. Rule (P_1) propagates

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

$$\frac{M_1 \rightsquigarrow [a]\ g \ \to \Sigma_{i\in I}\lambda_i : u_i \qquad M_2 \rightsquigarrow [a]\ g' \ \to \Sigma_{j\in J}\lambda'_j : u'_j \qquad a \in A}{M_1|[A]|M_2 \rightsquigarrow [a]\ g \wedge g' \ \to \Sigma_{i,j}\ \lambda_i * \lambda'_j : u_i \& u'_j}\ (\mathsf{P}_2)$$

Fig. 1. Semantics for PRISM networks

a command through parallel composition if α is empty or if the label a is not part of the set A. When the label a is in A, we apply rule (P_2). In this case, the product of the probabilities/rates must be taken by extending the two different branches to every possible combination. This also includes the combination of the associated assignments.

Based on the relation above, given $M \rightsquigarrow [\alpha]\ g \ \to \Sigma_{i\in I}\lambda_i : u_i$ and two states S and S', we define the function

$$\mu([\alpha]\ g \ \to \Sigma_{i\in I}\lambda_i : u_i,\ S,\ S'\ ) = \Sigma_{S[u_i]=S', i\in I}\lambda_i$$

which gives the probability/rate for the system to go from state S to state S' after executing command $[\alpha]\ g \ \to \Sigma_{i\in I}\lambda_i : u_i$, for some α. If the λ_i are probabilities, then the function must be a probability distribution. Note that $\mu(F, S, S')$ only denotes the probability/rate for the system to move from state S to state S' after executing command F. However, there can be other commands derived from a given network M through the relation $\rightsquigarrow$ that would cause a transition from S to S'. Therefore, we define the transition relation on states $M \vdash S \longrightarrow_\lambda S'$ as

$$\frac{\forall j, \alpha.\ M \rightsquigarrow F_j \qquad S \vdash F_j}{M \vdash S \longrightarrow_{\sum_j \mu(F_j,S,S')} S'}\ (\mathsf{Transition})$$

where $S \vdash [\alpha]\ g \ \to \Sigma_{i\in I}\lambda_i : u_i$ is defined as $g{\downarrow_S}$. Note that given the declarative aspect of the PRISM language, a given network M never changes while the state of the system evolves.

It is important to point out that, in general, the transition rule above does not give the exact probability of a transition in case of a Markov chain (DTMC), since the sum $\sum_j \mu(F_j, S, S')$ could be a value greater than 1. In order to get the right probability, the value has to be normalised for all reachable S'. In the next section, we will show that this is not an issue for networks that are obtained from our translation from choreography to PRISM.

Example 3. Consider the following network M:

$$\begin{array}{ll}
\mathtt{p} : \{ & [\,]\ x = 0 \ \rightarrow 1 : (x' = 1) \\
 & [a]\ y < 1 \ \rightarrow 0.4 : (x' = x + 1) \ + \ 0.6 : (x' = x) \quad \} \\
\mathtt{q} : \{ & [\,]\ y = 0 \ \rightarrow 1 : (y' = 1) \\
 & [a]\ x < 1 \ \rightarrow 0.5 : (y' = y + 1) \ + \ 0.5 : (y' = y) \quad \}
\end{array}$$

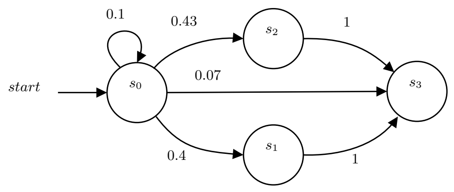
above, the two modules p and q can both do independent actions, as well as synchronising on label a. Applying the semantics, we can easily derive $M \rightsquigarrow [\,]\ x = 0 \ \rightarrow 1 : (x' = 1)$, $M \rightsquigarrow [\,]\ y = 0 \ \rightarrow 1 : (y' = 1)$, and $M \rightsquigarrow F$, such that

$$\begin{aligned}
F = [a]\ x < 1\ \&\ y < 1 \ \rightarrow \quad & 0.2 : (x' = x + 1)\ \&\ (y' = y + 1) \\
 + \ & 0.2 : (x' = x + 1)\ \&\ (y' = y) \\
 + \ & 0.3 : (x' = x)\ \&\ (y' = y + 1) \\
 + \ & 0.3 : (x' = x)\ \&\ (y' = y)
\end{aligned}$$

Let $s_0 = (x = 0, y = 0)$, $s_1 = (x = 1, y = 0)$, $s_2 = (x = 0, y = 1)$, and $s_0 = (x = 1, y = 1)$ be all the possible states, with s_0 a starting state. Then, we have that:

$$\begin{array}{ll}
\mu([\,]\ x = 0 \ \rightarrow 1 : (x' = 1), s_0, s_1) = 1 & \mu(F, s_0, s_1) = 0.2 \\
\mu([\,]\ y = 0 \ \rightarrow 1 : (y' = 1), s_0, s_2) = 1 & \mu(F, s_0, s_2) = 0.3 \\
\mu(F, s_0, s_0) = 0.3 & \mu(F, s_0, s_3) = 0.2
\end{array}$$

Now, by (Transition), we have that $M \vdash s_0 \rightarrow_{1.2} s_1$, $M \vdash s_0 \rightarrow_{1.3} s_1$. Clearly, both transitions should be normalised, finally yielding the following DTMC:



4 Projection

Our next task is to provide a mapping that can translate choreographies into the PRISM language. This section addresses projection formally.

Mapping Choreographies to PRISM. The operation of generating endpoint code from a choreography is known as *projection*. We observe that to simulate a choreography interaction in PRISM, we need to use labels on which each involved module can synchronise. Therefore, without loss of generality, we make a slight abuse of notation and assume that each interaction in a choreography is annotated with a unique label (which will be used by the projection). We call such a choreography an *annotated choreography*. For example, the choreography $\mathtt{p} \to \{\mathtt{q}\} + \ \lambda_1 : \mathbf{0}\ +\ \lambda_2 : \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathbf{0}$ is going to be annotated as $\mathtt{p} \to^a \{\mathtt{q}\} + \lambda_1 : \mathbf{0} + \lambda_2 : \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathbf{0}$. We now separate the definition of projection based on whether we are dealing with rates or probabilities. We start with choreographies with rates:

Definition 1 (Projection, CTMC). *Given an annotated choreography with rates C, a module $\mathtt{p}$, and a natural number ι, we define the function* proj *as:*

$$\mathsf{proj}(\mathsf{q}, \mathtt{p} \to^a \{\mathtt{p}_1, \ldots, \mathtt{p_n}\} : \Sigma_{j\in J}\lambda_j : u_j;\ C_j, \iota) = \qquad \boxed{\textit{if } \mathsf{q} = \mathtt{p}}$$
$$\Big\{[a_j]\ \ s_q = \iota\ \to \lambda_j :\ s_q = s_q + 1 + \textstyle\sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ u_j \downarrow_q \Big\}_{j\in J}$$
$$\cup\ \textstyle\bigcup_j \mathsf{proj}(\mathsf{q}, C_j, \iota + 1 + \sum_{k=1}^{j-1} \mathsf{nodes}(C_k))$$

$$\mathsf{proj}(\mathsf{q}, \mathtt{p} \to^a \{\mathtt{p}_1, \ldots, \mathtt{p_n}\} : \Sigma_{j\in J}\lambda_j : u_j;\ C_j, \iota) = \qquad \boxed{\textit{if } \mathsf{q} \in \{\mathtt{p}_1, \ldots, \mathtt{p}_n\}}$$
$$\Big\{[a_j]\ \ s_q = \iota\ \to 1 :\ s_q = s_q + 1 + \textstyle\sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ u_j \downarrow_q \Big\}_{j\in J}$$
$$\cup\ \textstyle\bigcup_j \mathsf{proj}(\mathsf{q}, C_j, \iota + 1 + \sum_{k=1}^{j-1} \mathsf{nodes}(C_k))$$

$$\mathsf{proj}(\mathsf{q}, \mathtt{p} \to^a \{\mathtt{p}_1, \ldots, \mathtt{p_n}\} : \Sigma_{j\in J}\lambda_j : u_j;\ C_j, \iota) =\ \mathsf{proj}(\mathsf{q}, C_1, \iota) \qquad \boxed{\textit{if } \mathsf{q} \notin \{\mathtt{p}, \mathtt{p}_1, \ldots, \mathtt{p}_n\}}$$
$$\textit{such that } \forall i, j.\ \mathsf{proj}(\mathsf{q}, C_i, \iota) = \mathsf{proj}(\mathsf{q}, C_j, \iota)$$

$$\mathsf{proj}(\mathsf{q}, \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \iota) = \qquad \boxed{\textit{if } \mathsf{q} = \mathtt{p}}$$
$$\left\{ \begin{array}{l} []\ \ s_q = \iota\ \&\ E\ \ \to\ \ 1 : s'_q = \iota + 1, \\ []\ \ s_q = \iota\ \&\ \mathsf{not}(E)\ \ \to\ \ 1 : s'_q = \iota + \mathsf{nodes}(C_1) + 1 \end{array} \right\}$$
$$\cup\ \ \mathsf{proj}(\mathsf{p}, C_1, \iota + 1)\ \ \cup\ \ \mathsf{proj}(\mathsf{p}, C_2, \iota + \mathsf{nodes}(C_1) + 1)$$

$$\mathsf{proj}(\mathsf{q}, \mathtt{if}\ E@\mathtt{p}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \iota) = \qquad \boxed{\textit{if } \mathsf{q} \neq \mathtt{p}}$$
$$\mathsf{proj}(\mathsf{q}, C_1, \iota + 1)\ \ \cup\ \ \mathsf{proj}(\mathsf{q}, C_2, \iota + \mathsf{nodes}(C_1) + 1)$$

$$\mathsf{proj}(\mathsf{q}, \mathbf{0}, \iota) = \emptyset$$

We go through the various cases of the definition above. The first three cases handle the projection of an interaction. If we project the first module $\mathtt{p}$, then we create one command per branch, assigning the corresponding rate. Note that this is possible since we are dealing with rates. The additional variable $s_\mathtt{q}$ is the counter for this module and we identify uniquely this interaction. In order to do it consistently for follow up statements of the subterm choreographies, we use the function $\mathsf{nodes}(C)$ which returns the number of nodes of C, i.e., the number of steps of the projection function. Obviously, when projecting the next branch we need to consider all other possible branches that may have been already projected. Intuitively, a label and an integer (ranged by ι) identify a node in the abstract syntax tree of a choreography. Also, note that we assume that from a label a we can generate distinct sublabels a_j just by adding some index j. For the sake of space, we do not define the function precisely, but we observe that this could also be easily defined via the label annotations. The second case defines the projection of an interaction when we are projecting one of the modules $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$. Similarly to the previous case, we define a command for each branch of the interaction. However, the rate of each command is going to be 1. The if-then-else construct is only interesting for module p: in this case, the module makes an internal choice based on the evaluation of the guard E. The other modules will be projected according to choreographies C_1 and C_2.

As hinted above, the projection in Definition 1 would be incorrect if instead of using rates we used probabilities. This is simply because we cannot force both p and $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$ to take the same branch with the probability distribution of the λ_i's. To fix this problem, we have the following definition instead:

Definition 2 (Projection, DTMC). *Given an annotated choreography with probabilities C, a module p, and a natural number ι, we define* proj *as:*

$$\mathsf{proj}(\mathsf{q}, \mathsf{p} \to \{\mathsf{p_1}, \ldots, \mathsf{p_n}\} : \Sigma_{j \in J} \lambda_j : u_j;\ C_j, \iota) = \qquad \boxed{\textit{if}\ \mathsf{q} = \mathsf{p}}$$

$$\begin{Bmatrix} [\,]\ s_q = \iota \ \to\ \sum_{j \in J} \lambda_j : s'_q = \iota + 1 + j, \\ \{[l_j]\ s_q = \iota + 1 + j \ \to\ 1 : s'_q = \iota + 1 + \sum_{k=1}^{j-1} \mathsf{nodes}(C_k)\ \&\ u_j \downarrow_q\}_{j \in J} \end{Bmatrix}$$
$$\cup\ \bigcup_j \mathsf{proj}(\mathsf{q}, C_j, s + 1 + \textstyle\sum_{k=1}^{j-1} \mathsf{nodes}(C_k))$$

The other cases of the definition are equivalent to those in Definition 1.

The fix is immediate: module p takes a (probabilistic) internal decision on the j^{th} branch and then synchronises on label l_j with $\{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$.

Correctness. Our projection operations are correct with respect to the semantics of choreographies and PRISM. In the sequel, S_+ is a state S extended with the extra variables s_q (for each module in a choreography) used by the projection. In the projection, we utilize alphabetized parallel composition $\|$, wherein modules synchronize solely on labels that appear in both modules.

Theorem 1 (EPP). *Given an annotated choreography C, we have that $(S, C) \longrightarrow_\lambda (S', C')$ iff $\|_\mathsf{q}\ \mathsf{proj}(\mathsf{q}, C, \iota) \vdash S_+ \longrightarrow_\lambda S'_+$.*

Proof (Sketch). The proof must be separated into whether we deal with rates and with probabilities. Let us focus on the projection in Definition 1. The proof proceeds by induction on the term C. For $C = \mathsf{p} \to \{\mathsf{p_1}, \ldots, \mathsf{p_n}\} : \Sigma_{j \in J} \lambda_j : u_j;\ C_j$, the key case, we clearly have that, for any state S there exists S' such that $(S, C) \longrightarrow_{\lambda_j} (S', C_j)$. We need to show two things: first, that the projection $\|_\mathsf{q}\ \mathsf{proj}(\mathsf{q}, C, \iota)$ of C can make the same transition; second, that if the projection makes a transition, it must be corresponding to that of the choreography above.

We now observe that this state of the CTMC in the translation is uniquely identified by the counter ι. Moreover, the uniqueness of the label a makes sure that all and only those modules involved in this interaction synchronise with this action (this shows from the rules in Figure 1). As a consequence, the commands generated by this step of the translation are such that any state S_+ is exclusively going to enable these commands (because of the guard $s_{\mathtt{q}} = \iota$) which obviously implies that it must be done with rate λ_j applying the rules in Figure 1. The same argument can be applied for the opposite direction. The other cases are similar. The case for DTMC is also similar.

□

5 Implementation

We implemented our language in 1246 lines of Java, by defining its grammar and using ANTLR [1] to generate both parser and visitor components. Each syntax node within the abstract syntax tree (AST) was encapsulated in a corresponding `Node` class, with methods within these classes used for PRISM code generation.

```
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 of the modules. New commands are appended to the set of commands for each respective module as they are generated.
- `index` and `maxIndex`: indices for tracking the current module being analyzed.
- `isCtmc`: a boolean flag indicating if a CTMC is being generated, crucial for projection generation logic.
- `labels`: existing labels; essential for checking label uniqueness.
- `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 needs 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 has as a field 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 efficiency of the projection.

One of the differences between formal syntax and implementation is the presence of module parameterizations in the latter. Specifically, to avoid repeating the same commands for each duplicated module, we utilize the notation `"[i]"` for each module with the same behaviour. This enables us to perform module renaming, a principle that also extends to variables and their updates. For instance, in Listing 1.2, the process `P` performs the same branch for each process

`Q[i]`. For every module in the system, where the index i ranges from 1 to n, there exists a corresponding PRISM module. For instance, for the example in Listing 1.2, if i ranges from 1 to 2, we will generate the respective PRISM code as depicted in Listing 1.3.

```
C ≔ P → Q[i] : (+["lambda1*mu[i]"] "(x'=1)" "(y[i]'=2)" ; C
                +["lambda2*mu[i]"] "(x'=3)" "(y[i]'=1)" ; C )
```

Listing 1.2. Example of an use of parameterization in the choreographic language

```
ctmc
module Q1
   Q1_STATE : [0..1] init 0;
   y1 : [0..N] init 0;
   [RLICV] (Q1_STATE=0) → mu1 : (y1'=2)&(Q1_STATE'=0);
   [OKAMT] (Q1_STATE=0) → mu1 : (y1'=1)&(Q1_STATE'=0);
endmodule

module Q2
   Q2_STATE : [0..1] init 0;
   y2 : [0..N] init 0;
   [OMPXG] (Q2_STATE=0) → mu2 : (y2'=2)&(Q2_STATE'=0);
   [AQNZR] (Q2_STATE=0) → mu2 : (y2'=1)&(Q2_STATE'=0);
endmodule

module P
   P_STATE : [0..2] init 0;
   x : [0..N] init 0;
   [RLICV] (P_STATE=0) → lambda1 : (x'=1)&(P_STATE'=0);
   [OKAMT] (P_STATE=0) → lambda2 : (x'=3)&(P_STATE'=0);
   [OMPXG] (P_STATE=0) → lambda1 : (x'=1)&(P_STATE'=0);
   [AQNZR] (P_STATE=0) → lambda2 : (x'=3)&(P_STATE'=0);
endmodule
```

Listing 1.3. PRISM code generated for the choreography in Listing 1.2

This modular approach systematically represents and integrates each system component within the PRISM framework, enabling comprehensive analysis and synthesis of the system's behavior. Importantly, these internal optimizations do not impact the projection process, as they focus on efficiency and code management rather than altering the overall structure or behavior of the projection.

The other differences are primarily syntactic. Updates of the same process are delineated by quotation marks, such as `"(x'=1)"`. Additionally, rates and probabilities are represented differently. In our choreographic language the rate/probability of interaction is represented as the product of rates/probabilities of each process. For example, in Listing 1.2, we use `lambda1*mu[i]` to indicate that the rate of the first process (`P`) is `lambda1`, while the rate of the second process (`Q[i]`) is represented by `mu[i]`. If multiple processes are interacting, the rate/probability is the product of all corresponding rates/probabilities (`lambda_1...*lambda_n`).

In our implementation, we ensure a single enabled action per state by enforcing label uniqueness and unique state-associated variables. This clarity aids in accurately determining enabled actions and enhances system reliability, facilitating analysis and comprehension of system dynamics.

6 Benchmarking

In this section, we report an experimental evaluation of our language. In order to show that using choreographies is beneficial in the presence of concurrency, we included an example modelling a single process. In particular, we focus on three benchmarks: the dice program that we compare with the test cases reported in the PRISM repository[4]; the Bitcoin Proof of Work protocol and the Hybrid Casper protocol, presented in [6, 11]. For lack of space, we do not report the generated PRISM files, they can be found in the online repository [3].

6.1 The Dice Program

The first test case is the Dice Program [15], a model involving only a single process. The following program simulates a die using only unbiased coins. Starting 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 particular, the PRISM model uses two variables: `d` which represents the value of the dice, ranging from 0 to 6; `STATE` which is the state variable, ranging from 0 to 7. Initially, both variables are set to 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)" ; 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 1.4. Choreography for the Dice Program

As expected, since there is no concurrency, there are no significant improvements to the code in using choreographies. The PRISM model produced by our method closely aligns with the original model with only minimal details that differ between the two. Additionally, to ensure the generated program's accuracy, we evaluated whether the probability of reaching a state where the dice displays `d=k` for every possible `k` from 1 to 6 equals 1/6. Our simulations shown that the probabilities calculated by our generated model fully matched those from the original PRISM model.

6.2 Proof of Work Bitcoin Protocol

In [6], the authors extended the PRISM model checker syntax to incorporate dynamic data types, enhancing its capabilities to model the Proof of Work protocol used in the Bitcoin blockchain [21].

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

```
{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 1.5. Choreography for the Proof of Work Bitcoin Protocol

In summary, the code depicts miners engaging in solving PoW, updating their ledgers, and communicating with the network. When synchronizing with the hasher, a miner tries to solve a cryptographic puzzle. Successful attempts result in adding a new block to its ledger and updating other miners' block sets via the network. Unsuccessful attempts involve extracting a block from its set, updating its ledger and block sets, and continuing the PoW process.

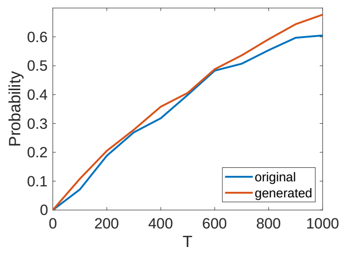


Fig. 2.

The PRISM model we created is more verbose than the one in [6], mainly because we consistently generate the else branch for if-then-else expressions, resulting in a higher number of instructions. Despite this, the experimental results for block creation probability (Figure 2) remain unaffected. Any discrepancies between the original and generated models are due to inherent variations in the simulation-based calculation of probability.

6.3 Hybrid Casper Protocol

We now present the Hybrid Casper Protocol [11]. The Hybrid Casper protocol represents a hybrid consensus protocol for blockchains, merging features from both Proof of Work and Proof of Stake protocols.

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*. Each *Validator* module closely resembles the *Miner* module from the previous protocol. The module *Vote_Manager* is responsible for storing maps containing votes for each block and computing associated rewards/penalties. The choreographic model for this example is reported in Listing 1.6. The code resembles that of the Proof of Work protocol, but with two key distinctions: *(i)* validators do not need to synchronize with hasher processes to solve the cryptopuzzle, and *(ii)* each validator can either create a new block, receive blocks from the network module, or determine if it's eligible to vote for specific blocks. For lack of space, we detailed only part of the code, the complete model can be found in [3].

```
{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{...}
  else{
    Validator[i] -> Vote_Manager :(["1*1"] "(Votes'=addVote(Votes,b[i],stake[i]))"; PoS)
  }
 +["hR*1"] ; if "!isEmpty(set[i])"@Validator[i] then { ... }
             else{ PoS }
 +["rC*1"] "(lastCheck[i]'=extractCheckpoint(listCheckpoints[i],lastCheck[i]))"...}
```

Listing 1.6. Excerpt of the Hybrid Casper Protocol as a choreography

The generated code is very similar the one outlined in [11], with the main distinction being the greater number of lines in our generated model. This difference is due to the fact that certain commands could be combined, but our generation lacks the automatic capability to perform this check. While the results obtained for the probability of creating a block reported in Figure 3 exhibit similarity, running simulations for the generated model takes PRISM 39.016 seconds, compared to the 22.051 seconds required for the original model.

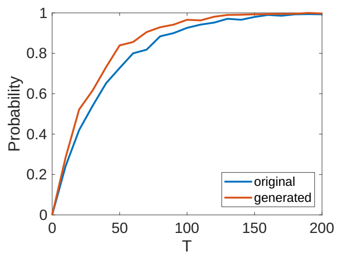


Fig. 3.

7 Related Work and Discussion

Related Work. Choreographic programming [20] is a language paradigm for specifying the expected interactions (communications) of a distributed system from a global viewpoint, from which decentralised implementations can be generated via projection. The notion of choreography has been substantially explored in the last decade, both from a theoretical perspective, e.g., [8, 9], to full integration into fully-fledged programming languages, such as WS-CDL [14] and Choral [12]. Nevertheless, there is a scarcity of research on probabilistic aspects of choreographic programming. To the best of our knowledge, Aman and Ciobanu [4, 5] are the only ones who studied the concept of choreography and probabilities. Their work augments multiparty session types (type abstractions for communicating systems that use the concept of choreography) with a probabilistic internal choice similar to the one used by our choreographic branching. However, they do not provide any semantics with state in terms of Markov chains, and, most importantly, they do not project into a probabilistic declarative language model such as PRISM. Carbone et al. [7] define a logic for expressing properties of a session-typed choreography language. However, the logic is undecidable and has no model-checking algorithm. As far as our knowledge extends, there is currently no work that generates probabilistic models from choreographic languages that can be then model-checked.

Discussion and Future Work. The ultimate goal of the proposed framework is to use the concept of choreographic programming to improve several aspects, including usability, correctness, and efficiency in modeling and analysing systems. In this paper, we address usability and efficiency of modelling systems by proposing a probabilistic choreography language. Our language improves the intuitive modeling of concurrent probabilistic systems. Traditional modeling languages often lack the expressive clarity needed to effectively capture the intricacies of such systems. By designing a language specific for choreographing system behaviors, we provide an intuitive means of specifying system dynamics. This approach enables a more natural and straightforward modeling process, essential for accurately representing real-world systems and ensuring the efficacy of subsequent analysis.

Although choreograhies and the projection function aim to abstract away low-level details, providing instead a higher-level representation of system behaviors, the choreographic approach can have some limitations in expressivity. Some of the case studies presented in the PRISM documentation [2] cannot be modeled by using our current language. Specifically, there are two main cases where our approach encounters limitations: *(i)* in the asynchronous leader election case study, our language prohibits the use of an 'if-then' statement without an accompanying 'else' to prevent deadlocked states; *(ii)* in probabilistic broadcast protocols or cyclic server polling system models, the system requires probabilistic branching to synchronize different modules based on the selected branch. These issues could be fixed by extending our choreographic language further and are therefore left as future work.

Additionally to these extensions, we conjecture that our semantics for choreographies may be used for improving performance by directly generating a CTMC or a DTMC from a choreography, bypassing the projection into the PRISM language. In fact, the Markov chain construction from a choreography seems to be faster than the construction from a corresponding projection in the PRISM language, since it is not necessary to take into account all the possible synchronisations in the rules from Fig. 1. A formal complexity analysis, an implementation, and performance benchmarking are left as future work.

In conclusion, this paper has introduced a framework for addressing the challenges of modelling and analysing concurrent probabilistic systems. The development of a choreographic language with tailored syntax and semantics offers an intuitive modeling approach. We have established the correctness of a projection function that translates choreographic models to PRISM-compatible formats. Additionally, our compiler enables seamless translation of choreographic models to PRISM, facilitating powerful analysis while maintaining expressive clarity. These contributions bridge the gap between high-level modeling and robust analysis in probabilistic systems, paving the way for advancements in the field.

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

References

1. ANTLR - another tool for language recognition. https://www.antlr.org/
2. Prism documentation. https://www.prismmodelchecker.org/, accessed: 2023-09-05
3. Repository. https://github.com/adeleveschetti/ChoreoPRISM, accessed: 2024-01-31
4. Aman, B., Ciobanu, G.: Probabilities in session types. In: Marin, M., Craciun, A. (eds.) Proceedings Third Symposium on Working Formal Methods, FROM 2019, Timişoara, Romania, 3-5 September 2019. EPTCS, vol. 303, pp. 92–106 (2019). https://doi.org/10.4204/EPTCS.303.7, https://doi.org/10.4204/EPTCS.303.7
5. Aman, B., Ciobanu, G.: Interval probability for sessions types. In: Ciabattoni, A., Pimentel, E., de Queiroz, R.J.G.B. (eds.) Logic, Language, Information, and Computation - 28th International Workshop, WoLLIC 2022, Iaşi, Romania, September 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13468, pp. 123–140. Springer (2022). https://doi.org/10.1007/978-3-031-15298-6_8, https://doi.org/10.1007/978-3-031-15298-6_8
6. 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
7. Carbone, M., Grohmann, D., Hildebrandt, T.T., López, H.A.: A logic for choreographies. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. EPTCS, vol. 69, pp. 29–43 (2010). https://doi.org/10.4204/EPTCS.69.3, https://doi.org/10.4204/EPTCS.69.3
8. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012). https://doi.org/10.1145/2220365.2220367, https://doi.org/10.1145/2220365.2220367
9. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 263–274. ACM (2013). https://doi.org/10.1145/2429069.2429101, https://doi.org/10.1145/2429069.2429101
10. Dannenberg, F., Kwiatkowska, M., Thachuk, C., Turberfield, A.: DNA walker circuits: computational potential, design, and verification. In: Soloveichik, D., Yurke, B. (eds.) Proc. 19th International Conference on DNA Computing and Molecular Programming (DNA 19). LNCS, vol. 8141, pp. 31–45. Springer (2013)
11. 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
12. Giallorenzo, S., Montesi, F., Peressotti, M.: Choral: Object-oriented choreographic programming. ACM Trans. Program. Lang. Syst. **46**(1), 1:1–1:59 (2024). https://doi.org/10.1145/3632398, https://dl.acm.org/doi/10.1145/3632398
13. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. In: Priami, C. (ed.) Proc.

Computational Methods in Systems Biology (CMSB'06). Lecture Notes in Bioinformatics, vol. 4210, pp. 32–47. Springer Verlag (2006)
14. Honda, K., Yoshida, N., Carbone, M.: Web services, mobile processes and types. Bull. EATCS **91**, 160–185 (2007)
15. Knuth, D., Yao, A.: Algorithms and Complexity: New Directions and Recent Results, chap. The complexity of nonuniform random number generation. Academic Press (1976)
16. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic verification of hermanâ€™s self-stabilisation algorithm. Formal Aspects of Computing **24**(4), 661–670 (2012)
17. Kwiatkowska, M., Norman, G., Parker, D., Vigliotti, M.: Probabilistic mobile ambients. Theoretical Computer Science **410**(12–13), 1272–1303 (2009)
18. Kwiatkowska, M., Norman, G., Segala, R.: Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In: Berry, G., Comon, H., Finkel, A. (eds.) Proc. 13th International Conference on Computer Aided Verification (CAV'01). LNCS, vol. 2102, pp. 194–206. Springer (2001)
19. Kwiatkowska, M., Norman, G., Sproston, J.: Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. Formal Aspects of Computing **14**(3), 295–318 (2003)
20. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023)
21. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf` (2008)
22. Norman, G., Shmatikov, V.: Analysis of probabilistic contract signing. Journal of Computer Security **14**(6), 561–589 (2006)