Data Petri Nets Meet Probabilistic Programming

Martin Kuhn[1], Joscha Grüger[1,2], Christoph Matheja[3], and Andrey Rivkin[3](✉)

[1] German Research Center for Artificial Intelligence (DFKI), SDS Branch Trier, Trier, Germany
martin.kuhn@dfki.de

[2] University of Trier, Trier, Germany
grueger@uni-trier.de

[3] Technical University of Denmark, Kgs. Lyngby, Denmark
{chmat,ariv}@dtu.dk

Abstract. Probabilistic programming (PP) is a programming paradigm that allows for writing statistical models like ordinary programs, performing simulations by running those programs, and analyzing and refining their statistical behavior using powerful inference engines. This paper takes a step towards leveraging PP for reasoning about data-aware processes. To this end, we present a systematic translation of Data Petri Nets (DPNs) into a model written in a PP language whose features are supported by most PP systems. We show that our translation is sound and provides statistical guarantees for simulating DPNs. Furthermore, we discuss how PP can be used for process mining tasks and report on a prototype implementation of our translation. We also discuss further analysis scenarios that could be easily approached based on the proposed translation and available PP tools.

1 Introduction

Data Petri nets (DPNs) [5,18] is a popular formalism for data-aware processes that is used in business process management (BPM) and process mining (PM) for various tasks including discovery [18], conformance checking [7,8,18], formal verification and correctness analysis [9,10]. Moreover, it has been shown in [6] that DPNs can formalize the integration of a meaningful subset of BPMN with DMN S-FEEL decision tables. Recent work also addresses stochastic [19] and uncertainty-related [8] aspects of DPNs.

Simulation for DPNs. One of the key techniques in the BPM and PM repertoires is *simulation*, which allows for flexible analyses, such as "what-if" analysis, that often cannot be addressed by formal verification or that touch upon non-functional aspects (e.g., time, costs, resources) that are not reflected in process models [26,27]. For DPNs, the prime application of simulation is the generation of synthetic event logs aiming at closing the gap between missing datasets needed for substantial evaluation of discovery and analysis techniques. By far, only [13] explicitly provides a DPN simulation engine grounded in DPN semantics which allows to perform randomised generation of fixed-length executions.

Probabilistic Programming (PP). [11,12] is a paradigm developed by the programming languages and machine learning communities to make statistical models and reasoning about them with Bayesian inference accessible to non-experts. The key idea is to represent statistical models as programs and leave the development of efficient simulation and inference engines to the language developers.

A probabilistic program can be thought of as an ordinary program with the ability to sample from probability distributions. Running such a program means performing stochastic simulation: a single program execution corresponds to a single simulation of the underlying model. Modern PP systems have two characteristic features: First, they support *conditioning* the possible executions (or *feasible simulations*) on observed evidence, e.g. to refine a synthetic model using real-world data or user knowledge. Second, they support *inference techniques* to compute or approximate the *probability distribution* modeled by a program.

This raises the question whether one can leverage the existing PP machinery for simulating DPNs instead of developing ad-hoc simulators.

Contributions and Outline. In this paper, we explore whether and how probabilistic programs are a suitable abstraction for simulating DPNs such that (1) the simulation process is based on a statistical model clearly defined as a probabilistic program and (2) simulation, event log generation, and further (statistical) analyses could benefit from using PP together with its sampling and inference capabilities (cf. [11,12]). Our main contributions can be summarized as follows:

- We formalize an execution semantics for DPNs with schedulers, which are used in discrete-event simulation to resolve non-determinism [17] (→ Sect. 3).
- We formalize the essence of many existing probabilistic programming languages (→ Sect. 4) and then develop a novel systematic encoding of DPNs with schedulers into a probabilistic programming language (→ Sect. 5).
- We show that our encoding is correct, i.e. our PP encoding of DPN produces exactly the runs of the encoded DPNs and preserves the probabilities of all simulated runs with respect to the scheduler (→ Sect. 5.3).
- We discuss how to leverage our encoding and inference engines provided by PP systems for various Process Mining tasks. (→ Sect. 6).
- We report on a proof-of-concept implementation of our encoding into the PP language webPPL [11] and discuss two case studies (→ Sect. 7).

Related Work. Multiple approaches exist that generate synthetic logs via model simulation for data-aware processes and that can be applied to DPNs. Given the known relation between colored Petri nets [14] and DPNs [5], one may use CPN Tools (https://cpntools.org/) to produce logs without noise or incompleteness [20]. As far as we know, CPN Tools is the only (non-commercial) tool in which one can explicitly define schedulers for simulation tasks.

Alternatively, one may leverage connections between BPMN 2.0 models and DPNs [6] to generate multi-perspective logs. For example, in [21] the authors rely on an executable BPMN semantics that supports data objects for driving exclusive choices. to generate random (multi-perspective) event longs. Similarly, [2] generates multi-perspective logs by running a randomized play-out game.

As far as we know, [13] proposes a simulation engine that directly implements the DPN execution semantics and which randomly fires enabled transitions. The random choices cannot be directly influenced, e.g. by changing the underlying scheduler. Moreover, any statistical guarantees about the generated logs provided by the engine – e.g., are the generated traces representative with respect to an underlying stochastic process? – are, at best, ad hoc. The same holds for the above works on the randomised log generation for BPMN.

One may also rely on studied relationships between business process models and discrete event simulation (DES) [24]. For example, [22] applies DES to BPMN 2.0 models to produce multi-perspective logs. While our work is grounded in Bayesian statistics, DES takes a frequentist approach to simulation where all "variables" are already known by the domain expert. We refer to [3] for a discussion of the advantages of Bayesian approaches compared to DES.

2 Preliminaries

Sequences. A finite *sequence* σ over a set S of length $|\sigma| = n \in \mathbb{N}$ is a function $\sigma : \{1, \ldots, n\} \to S$. We denote by ϵ the empty sequence of length $n = 0$. If $n > 0$ and $\sigma(i) = s_i$, for $1 \le i \le n$, we write $\sigma = s_1 \ldots s_n$. The set of all finite sequences over S is denoted by S^*. The *concatenation* $\sigma = \sigma_1\sigma_2$ of two sequences $\sigma_1, \sigma_2 \in S^*$ is given by $\sigma : \{1, \ldots, |\sigma_1| + |\sigma_2|\} \to S$, such that $\sigma(i) = \sigma_1(i)$ for $1 \le i \le |\sigma_1|$, and $\sigma(i) = \sigma_2(i - |\sigma_1|)$ for $|\sigma_1| + 1 \le i \le |\sigma_1| + |\sigma_2|$.

Probability Distributions. A (discrete) *subprobability distribution* over a non-empty, countable set X is a function $\mu\colon X \to [0,1]$ s.t. $\sum_{x \in X} \mu(x) \le 1$. We say that $\mu(x)$ is the probability assigned to $x \in X$ and call μ a *distribution* if $\sum_{x \in X} \mu(x) = 1$. We denote by $\mathit{Dist}(X)$ (resp. $\mathit{SubDist}(X)$) the set of all (sub)distributions over X, respectively. We consider a few examples:

1. The *Dirac distribution* $\delta_x\colon X \to [0,1]$ assigns probability 1 to a fixed element $x \in X$ and probability 0 to every other element of X.
2. For two distinct elements $x, y \in X$ and some $p \in [0,1]$, the *Bernoulli distribution* $B(p, x, y)\colon X \to [0,1]$ models a coin-flip with bias p and possible outcomes x and y: it assigns p to x, $(1-p)$ to y, and 0 otherwise.
3. The *uniform distribution* `unif(a,b)`$\colon \mathbb{N} \to [0,1]$ assigns probability $^1/_{(b-a+1)}$ to values in $[a\,..\,b]$; to every other natural number $r \in \mathbb{N}$, it assigns 0.
4. The function $^B/_2\colon \mathbb{Q} \to [0,1]$ given by $^B/_2(0) = {^B/_2}(1) = {^1/_4}$ and $^B/_2(x) = 0$ for all $x \in \mathbb{Q} \setminus \{0,1\}$ is a subdistribution in $\mathit{SubDist}(\mathbb{Q})$, which represents the Bernoulli distribution $B(^1/_2, 0, 1)$ scaled by $^1/_2$.

For a subdistribution $\mu \in \mathit{SubDist}(X)$, its *normalized distribution* is given by $\mathit{normalize}(\mu) = \frac{\mu}{\sum_{x \in X} \mu(x)}$. For example, normalizing $^B/_2$ yields the distribution $\mathit{normalize}(^B/_2) = \frac{^B/_2}{^1/_2} = B(^1/_2, 0, 1)$. As in standard probability theory, we assume $^0/_0 = 0$, i.e. if μ assigns probability 0 everywhere, so does $\mathit{normalize}(\mu)$.

3 Data Petri Nets with Probabilistic Schedulers

Data Petri nets (DPNs) [5,18] extend traditional place-transition nets with the possibility to manipulate scalar case variables, which are used to constrain the evolution of the process through *guards* assigned to transitions. Each guard is split into a pre- and postcondition that is defined over two variable sets, V and V', where V is the set of case variables and V' keeps their primed copies used to describe variable updates.[1] We denote by $\mathcal{E}(X, \varDelta)$ the set of all *boolean* expressions over variables in X and constants in $\varDelta$. For an expression e, $V(e)$ and $V'(e)$ denote the sets of all unprimed and primed variables in e, respectively.

Definition 1 (Data Petri net). *A* data Petri net *(DPN) is a tuple* $N = \langle P, T, F, l, A, V, \varDelta, \mathsf{pre}, \mathsf{post}\rangle$*, where: (i)* P *and* T *are finite, disjoint sets of* places *and* transitions*, respectively; (ii)* $F \subseteq (P \times T) \cup (T \times P)$ *is a* flow relation*; (iii)* $l : T \rightarrow A$ *is a* labeling *function assigning activity names from* A *to every transition* $t \in T$*; (iv)* V *is a set of* case variables *and* $\varDelta$ *is a* data domain*; (v)* $\mathsf{pre} : T \rightarrow \mathcal{E}(V, \varDelta)$ *is a transition* precondition-assignment *function; and (vi)* $\mathsf{post} : T \rightarrow \mathcal{E}(V \cup V', \varDelta)$ *is a transition* postcondition-assignment *function.*

Given a DPN $N = \langle P, T, F, l, A, V, \varDelta, \mathsf{pre}, \mathsf{post}\rangle$, we will from now on write P_N, T_N, etc. to denote N's components; we omit the subscript if the given net is clear from the context. Given a place or transition $x \in (P_N \cup T_N)$ of N, the *preset* ${}^\bullet x$ and the *postset* $x^\bullet$ are given by ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet := \{y \mid (x, y) \in F\}$.

We next turn to the DPN execution semantics.

A *state* of a DPN N is a pair (M, α), where *(i)* $M : P_N \rightarrow \mathbb{N}$ is a total *marking* function, assigning a number $M(p)$ of *tokens* to every place $p \in P_N$ and *(ii)* $\alpha : V_N \rightarrow \varDelta$ is a total *variable valuation* function assigning a value to every variable in V_N. A DPN moves between states by firing (enabled) transitions.

Definition 2 (Transition enabling, firing). *Let* N *be a DPN,* (M, α) *a state,* $t \in T_N$ *a transition, and* $\beta : (V_N \cup V'_N) \nrightarrow \varDelta_N$ *be a partial valuation function.*

We denote by $(M, \alpha)[(t, \beta)\rangle$ *that the* step (t, β) *is* enabled *in* (M, α)*, i.e.*

(i) β *is defined for all the variables in* $\mathsf{pre}(t)$ *and* $\mathsf{post}(t)$ *and no other variables;*
(ii) for every $v \in V_N(\mathsf{pre}_N(t) \wedge \mathsf{post}_N(t))$*, we have that* $\beta(v) = \alpha(v)$*, i.e.,* β *matches* α *on all the (non-primed) variables of* t *that are not being modified;*
(iii) for every $p \in {}^\bullet t$*, we have* $M(p) \geq 1$*;*
(iv) $\beta \models \mathsf{pre}_N(t) \wedge \mathsf{post}_N(t)$*, i.e.,* β *satisfies the pre- and post-conditions of* t*.*

Moreover, we denote by $(M, \alpha)[(t, \beta)\rangle(M', \alpha')$ *that the state* (M', α') *is the result of* firing *transition* t *according to step* (t, β)*, i.e.* $(M, \alpha)[(t, \beta)\rangle$ *holds and (1) for every* $p \in P_N$*, we have* $M'(p) = M(p) - F_N(p, t) + F_N(t, p)$*; (2)* $\alpha'(v) = \beta(v')$ *for all* $v' \in V'_N(\mathsf{post}(t))$*; and (3)* $\alpha'(v) = \alpha(v)$ *for all* $v \in V_N(\mathsf{pre}_N(t) \wedge \mathsf{post}_N(t))$*.*

A *run* of a DPN N is a sequence of steps $\sigma = (t_1, \beta_1) \dots (t_k, \beta_k)$, where $k \in \mathbb{N}$. A state (M', α') is *reachable* from (M, α), if there is a run σ as

[1] Hereinafter, we assume that X' defines a copy of the set X in which each element $x \in X$ is primed. The same holds for individual elements.

above such that $(M,\alpha)[(t_1,\beta_1)\rangle(M_1,\alpha_1)[(t_2,\beta_2)\rangle\cdots[(t_k,\beta_k)\rangle(M',\alpha')$, denoted $(M,\alpha)[\sigma\rangle(M',\alpha')$. We denote by $\mathcal{S}_N$ the set of all states of N. A run σ is *legal* iff there are two states $(M,\alpha),(M',\alpha')\in\mathcal{S}_N$ s.t. $(M,\alpha)[\sigma\rangle(M',\alpha')$.

Simulating a DPN corresponds to generating runs. A crucial part of the simulation process is how to resolve non-deterministic choices attributed to multiple, simultaneously enabled transitions and possibly infinitely many partial valuations the transitions' pre- and post-conditions. This issue has been resolved in discrete event system simulation using schedulers that handle non-deterministic choices [17]. In the same vein, we define randomized schedulers for DPNs.

Definition 3 (DPN Scheduler). *A* scheduler *of a DPN N is a function*

$$\mathfrak{S}\colon \mathcal{S}_N \to \mathit{Dist}(T_N)\times(V'_N\to\mathit{Dist}(\Delta_N)).$$

In words, a scheduler $\mathfrak{S}$ assigns to every state (M,α) a probability distribution $\mathfrak{S}_T$ over the net's transitions and a mapping $\mathfrak{S}_V$ from the net's (primed) variables to probability distributions over domain values. Intuitively, $\mathfrak{S}_T$ resolves the nondeterminism that arises if multiple transitions are enabled: if all transitions are enabled, the probability of picking transition t is given by $\mathfrak{S}_T(t)$; if some transitions are not enabled, their probability is uniformly distributed amongst the enabled transitions. The function $\mathfrak{S}_V$ resolves for every $v'\in V'_N$ the nondeterminism that arises from postconditions that hold for different evaluations of v': the probability of assigning $\mathsf{d}\in\Delta_N$ to v' is given by $\mathfrak{S}_V(v')(\mathsf{d})$. We will modify this probability by conditioning on the fact that a transition's postcondition must hold after firing it. Notice that postconditions may introduce dependencies between case variables, even though their values are sampled independently.

Example 1. Consider the DPN N in Fig. 1 representing a simple auction process, where the last offer is stored in o and the time progression is captured by t. One possible scheduler uniformly selects an enabled transition, i.e. $\mathfrak{S}_T^{auc}=\texttt{unif}(1,|T_N|)$. Alternatively, one can model priorities between potentially simultaneously enabled transitions by selecting a different distribution that depends, e.g. on the value of the timer. Examples of distributions for selecting values for the variables t' and o', include the Poisson (over countable sets of timestamps and outcomes), uniform (over finite subsets of timestamps and outcomes such as $\mathfrak{S}_V^{auc}(t')=\texttt{unif}(0,99)$ and $\mathfrak{S}_V^{auc}(o')=\texttt{unif}(1,30)$), geometric, and normal distribution.[2] In the following examples, we will use $\mathfrak{S}^{auc}=(\mathfrak{S}_T^{auc},\mathfrak{S}_V^{auc})$ as the default scheduler. △

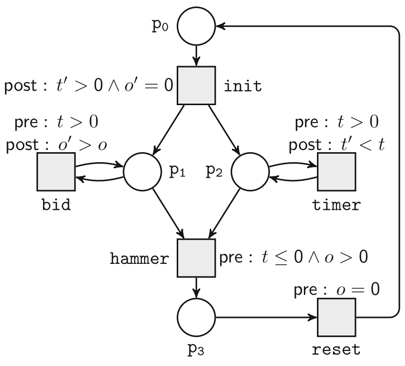

Fig. 1. A simple auction process [10]

[2] To improve readability, our approach is formalized for discrete distributions. However, continuous distributions are supported by many PP systems.

We will now quantify the probability of a run in terms of the likelihood[3] of each involved step. To this end, let $\mathsf{frontier}(M, \alpha) = \{t \mid \exists \beta \colon (M, \alpha)[(t, \beta)\rangle\}$ be the set of transitions that are enabled in a given state. Moreover, given a Boolean proposition p, we denote by $[p]$ its indicator function, i.e. $[p] = 1$ if p is $\top$ (true), and 0 if p is $\bot$ (false).

Definition 4 (Step Likelihood). *Given a net N and a scheduler $\mathfrak{S}$ with $\mathfrak{S}(M, \alpha) = (\mathfrak{S}_T, \mathfrak{S}_V)$, the* likelihood *of a step (t, β) in a state (M, α) is*

$$\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(t, \beta) \quad = \quad \underbrace{\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(t)}_{\textit{likelihood of selecting } t} \quad \cdot \quad \underbrace{\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(\beta \mid t)}_{\textit{likelihood of selecting } \beta \textit{ given } t} \quad ,$$

where the likelihood of selecting transition t among the enabled ones is

$$\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(t) \quad = \quad \frac{[t \in \mathit{frontier}(M, \alpha)] \cdot \mathfrak{S}_T(t)}{\sum\limits_{t' \in T_N} [t' \in \mathit{frontier}(M, \alpha)] \cdot \mathfrak{S}_T(t')}$$

and the likelihood of selecting $\beta \colon (V_N \cup V'_N) \nrightarrow \Delta_N$ given transition t s.t. $\beta(v) = \alpha(v)$, for every $v \in V_N(\mathit{pre}_N(t) \wedge \mathit{post}_N(t))$, is

$$\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(\beta \mid t) \quad = \quad [\beta \models \mathit{post}_N(t)] \cdot \prod_{x' \in V'_N(\beta)} \mathfrak{S}_V(x')(\beta(x')).$$

In the spirit of Bayes' rule, the likelihood of a step (t, β) is the likelihood that the scheduler selects transition t multiplied with the likelihood of selecting valuation β given the previous selection of t. Notice that $\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(t, \beta)$ is 0 if the chosen transition t is not enabled – and thus $[t \in \mathsf{frontier}(M, \alpha)] = 0$ – or if the chosen β does not satisfy t's postcondition – and thus $[\beta \models \mathsf{post}_N(t)] = 0$.

Example 2. Consider the DPN and scheduler from Example 1. The likelihood of the step $(\mathtt{timer}, \beta)$, where $\beta = \{t \mapsto 20, o \mapsto 5, t' \mapsto 11\}$, in state (M, α), where $M(\mathtt{p_1}) = M(\mathtt{p_2}) = 1$ and $M(\mathtt{p_0}) = M(\mathtt{p_3}) = 0$ and $\alpha = \{t \mapsto 20, o \mapsto 5\}$, is $\mathbb{P}_{\mathfrak{S}^{auc}}[M, \alpha]\,(\mathtt{timer}, \beta) = \frac{1/5}{3/5} \cdot 1/100$. △

We expand the above likelihood measure to DPN runs that reach a possibly infinite set $\mathcal{G}$ of *goal states* for the first time. Examples of goal states include all states with a final marking or all states in which $x \geq 15$. To this end, $\mathsf{Runs}(M, \alpha, \mathcal{G})$ denotes the set of all runs $\sigma = (t_1, \beta_1) \ldots (t_n, \beta_n)$, for all $n \geq 0$, such that, for all $k \in \{1, \ldots, n-1\}$,

$$\text{if} \quad (M, \alpha)[(t_1, \beta_1) \ldots (t_k, \beta_k)\rangle(M_k, \alpha_k) \quad \text{then} \quad (M_k, \alpha_k) \notin \mathcal{G}.$$

Notice that not all runs in $\mathsf{Runs}(M, \alpha, \mathcal{G})$ are legal and only some *may* reach states from $\mathcal{G}$. We then define the likelihood that a run $\sigma \in \mathsf{Runs}(M, \alpha, \mathcal{G})$ reaches $\mathcal{G}$ from (M, α) via its step likelihoods, or 0 if no goal state is ever reached.

[3] We use "likelihood" to refer to an "unnormalized probability", i.e. a value that is obtained from a subdistribution. The sum of all likelihoods may thus be less than 1.

Definition 5 (Likelihood of a Run). *Given a DPN N, a scheduler $\mathfrak{S}$, a state (M, α), goal states $\mathcal{G}$, and a run $\sigma \in \mathsf{Runs}(M, \alpha, \mathcal{G})$ over N, the* likelihood $\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(\sigma \models \Diamond\mathcal{G})$[4] *that σ reaches $\mathcal{G}$ from (M, α) is defined recursively as*

$$\begin{cases} 1, & \text{if } (M,\alpha) \in \mathcal{G} \\ \mathbb{P}_{\mathfrak{S}}[M,\alpha]\,(t,\beta) \cdot \mathbb{P}_{\mathfrak{S}}[M',\alpha']\,(\sigma' \models \Diamond\mathcal{G})\,, & \text{if } \sigma = (t,\beta)\,\sigma' \text{ and } (M,\alpha)[(t,\beta)\rangle(M',\alpha') \\ 0, & \text{otherwise.} \end{cases}$$

Technically, the likelihood of a run is defined analogously to the reachability probability of a trace in an (infinite-state) Markov chain – a well-established stochastic model for describing sequences of events (cf. [1,23]). The probability of the next step may thus depend on the *current* state but is independent of previously visted states. While this assumption is common and allows for dependencies between cases, e.g. case variables can be accessed and modified by different transitions, it can be seen as a limitation because those dependencies must be modeled explicitly in the DPN's state space.

Example 3. Consider the DPN from Example 1, set of goal states $\mathcal{G} = \{(M, \alpha) \in \mathcal{S}_N \mid M(\mathtt{p}_3) = 1\}$ and a run $\sigma = (\mathtt{init}, \{t' \mapsto 10\})(\mathtt{bid}, \{t \mapsto 10, o' \mapsto 5\})(\mathtt{timer}, \{t \mapsto 10, t' \mapsto 0\})(\mathtt{hammer}, \{t \mapsto 0, o \mapsto 5\})$. Then the likelihood to reach one of the goal states from initial state $(M_0, \alpha_0) = (\{\mathtt{p}_0 \mapsto 1, \mathtt{p}_1, \mathtt{p}_2, \mathtt{p}_3 \mapsto 0\}, \{t \mapsto 0, o \mapsto 0\})$ is $\mathbb{P}_{\mathfrak{S}^{auc}}[M_0, \alpha_0]\,(\sigma \models \Diamond\mathcal{G}) = (1 \cdot 1/100) \cdot (1 \cdot 1/10) \cdot (1/2 \cdot 1/100) \cdot (1/3 \cdot 1)$. △

To obtain the *probability* of a run, we then normalize the above likelihood with the likelihoods of all runs that can only reach $\mathcal{G}$ after firing all of their steps.

Definition 6 (Probability of a Run). *Given a DPN N, a scheduler $\mathfrak{S}$, a state (M, α), goal states $\mathcal{G}$ and a run σ over N, the* probability that σ reaches $\mathcal{G}$ *given that all runs starting in (M, α) eventually reach $\mathcal{G}$ is defined as*

$$\mathbb{P}_{\mathfrak{S}}[M,\alpha]\,(\sigma \mid \Diamond\mathcal{G}) \;=\; \frac{\mathbb{P}_{\mathfrak{S}}[M,\alpha]\,(\sigma \models \Diamond\mathcal{G})}{\sum_{\sigma' \in \mathsf{Runs}(M,\alpha,\mathcal{G})} \mathbb{P}_{\mathfrak{S}}[M,\alpha]\,(\sigma' \models \Diamond\mathcal{G})}.$$

Notice that a run σ is legal if $\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(\sigma \mid \Diamond\mathcal{G}) > 0$ holds, because the likelihood is set to 0 if we attempt to fire a transition that is not enabled or select a valuation that does not satisfy the transition's postcondition. Moreover, one can easily extract sequences of events, i.e. traces from a legal run and its initial state.

Our translation to probabilistic programs in Sect. 5 will make sure that, given a scheduler $\mathfrak{S}$ and a set of goal states $\mathcal{G}$, every run σ is produced with the probability $\mathbb{P}_{\mathfrak{S}}[M, \alpha]\,(\sigma \mid \Diamond\mathcal{G})$.

4 The Probabilistic Programming Language PPL

In this section, we give the necessary background on probabilistic programming by introducing a small probabilistic programming language called PPL whose features are supported by many existing PP systems (cf. [11,12]).

[4] As in [1], we use an LTL-like notation to denote that σ ends in a state in $\mathcal{G}$.

$$
\begin{aligned}
C \; ::= \; & x \; \texttt{:=} \; D && \text{(probabilistic assignment)} \\
\mid \; & \texttt{observe} \; B && \text{(condition on event } B\text{)} \\
\mid \; & \texttt{log} \; msg && \text{(add } msg \text{ to the log)} \\
\mid \; & C; C && \text{(sequential composition)} \\
\mid \; & \texttt{do} \; GC \; \texttt{od} && \text{(unbounded loop)} \\
\mid \; & \texttt{if} \; GC \; \texttt{fi} && \text{(conditional)} \\
GC \; ::= \; & B \xrightarrow{E} C && \text{(guarded command)} \\
\mid \; & GC \; \texttt{[]} \; GC && \text{(choice)}
\end{aligned}
$$

Fig. 2. Syntax of PPL.

```
x := uniform(1, 3);
if
      x = 1  --1/3-->  y := 4
  []  x > 1  --1/3-->  y := 5
  []  x > 1  --1/3-->  y := x + 2
fi;
observe B
```

Fig. 3. Example program.

Definition 7 (Syntax of PPL). *The set of* commands C *and* guarded commands GC *written in* PPL *is given by the grammar in Fig.* 2*, where* x *is a* program variable *taken from a finite set* $\textbf{\textit{Var}}$*,* D *is a* distribution expression *over* $\textbf{\textit{Var}}$*,* B *is an* Boolean expression *over* $\textbf{\textit{Var}}$*,* msg *is a* message[5] *over* $\textbf{\textit{Var}}$ *and* E *is a (rational)* probability expression *over* $\textbf{\textit{Var}}$*.*

Before we formalize the details, we briefly go over the intuitive meaning of each command. The *probabilistic assignment* x `:=` D samples a value from D and assigns the result to variable x. The *conditioning command* `observe` B checks whether B holds and proceeds if the answer is yes. Otherwise, the current execution is discarded as if it never happened.[6] The command `log` msg writes the value of msg to the program's log – an append-only list of messages, which we will use to output runs. The sequential composition $C_1; C_2$ first executes C_1, followed by C_2. The loop `do` GC `od` executes the guarded command GC until no guard in GC is enabled anymore. More precisely, consider the loop

$$\texttt{do}\; B_1 \xrightarrow{E_1} C_1 \;\texttt{[]}\; \ldots \;\texttt{[]}\; B_n \xrightarrow{E_n} C_n \;\texttt{od}.$$

If none of the guards $B_1, \ldots, B_n$ hold, the loop terminates. Otherwise, the loop randomly executes a command C_i whose guard holds, and repeats. The probability of executing a command is determined by the values of the expressions $E_1, \ldots, E_n$.[7] That is, if guard B_i holds and m is the total number of guards that currently hold, then the command C_i is executed with probability $E_i \cdot {}^{n}/_{m}$. Similarly to loops, the conditional `if` GC `fi` randomly executes one of the commands in GC whose guard holds, but terminates afterwards.

[5] Think: some string. In our DPN encoding, messages will correspond to steps (t, β).

[6] One can think of probabilistic programs as a stochastic simulator. Failing an observation then means that the simulation encountered an unrealistic result that should not be included in the result. A naive way to achieve this, is rejection sampling: discard the result and attempt to obtain another sample by restarting the simulation from scratch. More efficient approaches are discussed in [25].

[7] To ensure well-definedness, we assume that the values of $E_1, \ldots, E_n$ sum up to ≤ 1.

Example 4. Consider the PPL program C in Fig. 3. C first randomly assigns to x a value between 1 and 3 with probability $1/3$ each. If $x = 1$, it always assigns 4 to y. Otherwise, it either assigns 5 or $x + 2$ to y. Since only two out of three guards hold for $x > 1$, the probability of executing each assignment is $\frac{1/3}{2/3} = 1/2$.

If, for the moment, we ignore the `observe` B, then the probability of terminating with $y = 4$ is $1/3 + 1/3 \cdot 1/2 = 1/2$.

Analogously, the probability to stop with $y = 5$ is $2/3 \cdot 1/2 + 1/3 \cdot 1/2 = 1/2$.

How does conditioning on B affect those probabilities? For $B = (y = 5)$, we discard all executions except those that stop with $y = 5$. Hence, the probability to stop with $y = 5$ is 1, and 0 for any other value of y. For $B = (x > 1)$, we consider only those executions that assign 2 or 3 to x. Both assignments happen now with probability $1/2$. Hence, $y := 4$ is never executed. The probability of stopping with $y = 4$ changes to $1/2 \cdot 1/2 = 1/4$, i.e. the probability of assigning 2 to x and executing the last assignment. Analogously, the probability of stopping with $y = 5$ changes to $3/4$. For $B = (x = 1 \wedge y = 5)$, there is no feasible execution. Hence, we obtain a *sub*distribution that is zero for every value. $\triangle$

To assign formal semantics to PPL programs, we first define program states and discuss how expressions are evaluated.

States and Expressions. The set $\mathbf{PS} = \{ s \mid s \colon \mathbf{Var} \to \mathbb{Q} \}$ of *program states* consists of all assignments of rational numbers to program variables. Furthermore, the set of *program logs* $\mathbf{PL} = \{\ell \mid \ell \in \mathbf{Msg}^*\}$, where $\mathbf{Msg}$ is an infinite set of *messages* of interest, e.g. the set of all steps of a DPN.

We denote by $\mathbf{PSL} = \mathbf{PS} \times \mathbf{PL}$ the set of all pairs of a state s and a program log ℓ. We abstract from concrete syntax for expressions. Instead, we assume that, for every state, *distribution expressions* D evaluate to distributions over rationals, *Boolean expressions* B evaluate to $\mathbb{B} = \{\top, \bot\}$, *messages msg* evaluate to $\mathbf{Msg}$, and expressions E evaluate to rationals in $[0, 1]$, respectively. Formally, we assume the following evaluation functions for those expressions:

$$[\![B]\!] \colon \mathbf{PS} \to \mathbb{B} \quad [\![D]\!] \colon \mathbf{PS} \to \mathit{Dist}(\mathbb{Q}) \quad [\![\mathit{msg}]\!] \colon \mathbf{PS} \to \mathbf{Msg} \quad [\![E]\!] \colon \mathbf{PS} \to [0, 1] \cap \mathbb{Q}$$

Throughout this paper, we use common expressions, such as $x + y < 17$, $[x > 0]$, and `uniform(1,x+2)`, where the corresponding evaluation functions are straightforward, e.g. $\lambda s.s(x) + s(y) < 17$, $\lambda s.[s(x) > 0]$, and $\lambda s.\mathtt{uniform}(1, s(x) + 2)$.[8]

Semantics. A standard approach to assign (denotational) semantics to ordinary programs is to view them as a state transformer $[\![C]\!] \colon \mathbf{PS} \to 2^{\mathbf{PS}}$. That is, C produces a set of output states for any given input state. The set of output states can be empty, e.g. if the program enters an infinite loop.

For probabilistic programs, we obtain a more fine-grained view [15], because the probability of every output state can be quantified. Similarly, the semantics of PPL commands C is a function $\mathcal{S}[\![C]\!] \colon \mathbf{PSL} \to \mathit{SubDist}(\mathbf{PSL})$, that maps every initial state to a *subdistribution over output states.* We consider *sub*distributions, because we may lose probability mass, e.g. due to nontermination.

[8] We use lambda-notation, e.g., $\lambda s.1 + s(x)$, to define functions that depend on s.

Program C **(resp.** GC**)**	$[\![C]\!](s,\ell)$ **(resp.** $[\![GC]\!](s,\ell)$**)**
x `:=` D	$\sum_{q\in\mathbb{Q}}[\![D]\!](s)(q)\cdot\delta_{(s[x\mapsto q],\ell)}$
`log` msg	$\delta_{(s,\ell\,[\![msg]\!](s))}$
`observe` B	$[\![[B]]\!](s)\cdot\delta_{(s,\ell)}$
$C_1; C_2$	$\sum_{(s',\ell')\in\mathbf{PSL}}[\![C_1]\!](s,\ell)(s',\ell')\cdot[\![C_2]\!](s',\ell')$
`if` GC `fi`	$\frac{[\![\mathsf{guards}(GC)]\!](s)}{[\![\mathsf{weight}(GC)]\!](s)}\cdot[\![GC]\!](s,\ell)$
GC_1 `[]` GC_2	$[\![GC_1]\!](s,\ell)+[\![GC_2]\!](s,\ell)$
$B\xrightarrow{E}C_1$	$[\![[B]]\!](s)\cdot[\![E]\!](s)\cdot[\![C_1]\!](s,\ell)$
`do` GC `od`	$\lim_{n\to\infty}[\![C_n]\!](s,\ell)$, where $[\![C_0]\!](s,\ell)\ =\ \lambda(s',\ell').\,0$ $[\![C_{n+1}]\!](s,\ell)\ =\ (1-[\![\mathsf{guards}(GC)]\!](s))\cdot\delta_{(s,\ell)}+[\![\texttt{if}\ GC\ \texttt{fi};C_n]\!](s,\ell)$

Fig. 4. Semantics of PPL programs. Here, $(s,\ell)\in\mathbf{PSL}$. We denote by $s[x\mapsto v]$ the update of s in which the value of x is set to v, i.e. $s[x\mapsto v](y)=v$ if $y=x$, and $s[x\mapsto v](y)=s(y)$, otherwise. $\ell[\![msg]\!](s)$ denotes the concatenation of log ℓ and the evaluation of message msg in s. $\mathsf{guards}(GC)$ and $\mathsf{weight}(GC)$ are defined further below.

We formalize $\mathcal{S}[\![C]\!]$ in two steps: We first define a function $[\![C]\!]$ that treats observation failures like nontermination, i.e. failing an observation loses probability mass. After that, we *normalize* $[\![C]\!]$ by redistributing the lost probability mass among the feasible executions. Formally:

Definition 8 (Semantics of PPL**).** *The (sub-)distribution $\mathcal{S}[\![C]\!](s,\ell)$ computed by* PPL *program C for initial state-log pair (s,ℓ) is defined as*

$$\mathcal{S}[\![C]\!](s,\ell)\ =\ \mathit{normalize}([\![C]\!](s,\ell))\ =\ \frac{[\![C]\!](s,\ell)}{\sum_{(s',\ell')\in\boldsymbol{PSL}}[\![C]\!](s,\ell)(s',\ell')},$$

where the (unnormalized) transformer $[\![C]\!]\colon\boldsymbol{PSL}\to\mathit{SubDist}(\boldsymbol{PSL})$ is defined inductively on the structure of (guarded) commands in Fig. 4.

Intuitively, $\mathcal{S}[\![C]\!](s,\ell)(s',\ell')$ is the probability that executing PPL program C on initial program state s and log ℓ terminates in program state s' with log ℓ'.

While our semantics is precise, we remark that there also exist "sampling-based semantics" for probabilistic programs in the literature, e.g. [4], which guarantee that, given enough samples, the computed (sub)distribution will converge towards the subdistribution $[\![C]\!](s,\ell)$. Depending on the chosen inference engine applied, we thus either get exact or approximate guarantees.

Definition of $[\![C]\!]$. We now go over the definition of $[\![C]\!]$ in Fig. 4 and formal notation that is not explained in the caption. In each case, we are given an initial state-log pair $(s,\ell)\in\mathbf{PSL}$ and have to produce a final subdistribution $[\![C]\!](s,\ell)\in\mathit{SubDist}(\mathbf{PSL})$ depending on program C.

For the *probabilistic assignment* x `:=` D, initial state s and log ℓ, we sample a rational value q from the distribution $[\![D]\!](s)\in\mathit{Dist}(\mathbb{Q})$ and assign that value to variable x – the program state is thus updated to $s[x \mapsto q]$. To compute the final subdistribution, we sum over all possible samples $q \in \mathbb{Q}$, weighing each sample by its probability $[\![D]\!](s)(q)$, and require that the final state is the updated one using the Dirac distribution $\delta_{(s[x \mapsto q], \ell)}$. Notice that x `:=` D behaves like a standard assignment if $[\![D]\!](s)$ is a Dirac distribution (see [16]).

For the command `log` *msg*, we evaluate the message *msg* in the current and append the result to the current program log. The final subdistribution is thus the Dirac distribution wrt. the current program state and the updated log.

For the *conditioning command* `observe` B, the initial state s and log ℓ remain unchanged if B holds in s – we return the Dirac distribution $\delta_{(s,\ell)}$. Otherwise, we lose all probability mass and the final subdistribution is $\lambda(s', \ell').0$.[9] Formally, we denote by $[B]$ the indicator function of B (i.e., $[\![[B]]\!](s) = 1$ if $[\![B]\!](s) = \top$; otherwise, $[B](s) = 0$). The semantics of `observe` B is then $[\![[B]]\!](s) \cdot \delta_{(s,\ell)}$.

For the *sequential composition* $C_1; C_2$, we return the subdistribution obtained from running C_2 on every state-log pair (s', ℓ') weighted by the likelihood that executing C_1 on the initial state-log pair (s, ℓ) terminates in (s', ℓ').

The semantics of conditions `if` GC `fi` and loops `do` GC `od` depends on the guarded command GC, which we consider first. If GC is a branch of the form $B \xrightarrow{E} C$, then C can be executed with probability E if B holds. The resulting subdistribution is thus $[B] \cdot E \cdot \mathcal{S}[\![C]\!](\mu)$. If GC is a choice GC_1 `[]` GC_2, we sum the subdistributions computed for C_1 and C_2.

The semantics of `if` GC `fi` executes some branch in GC whose guard is enabled and normalizes among the enabled branches. That is, it uniformly distributes the probabilities of branches whose guard is not enabled among the branches whose guard is enabled. To formalize this behaviour, we use two auxiliary definitions. First, $\mathsf{guards}(GC)$ counts how many guards in GC hold:

$$\mathsf{guards}(GC) = \begin{cases} [B], & \text{if } GC = (B \xrightarrow{E} C) \\ \mathsf{guards}(GC_1) + \mathsf{guards}(GC_2) & \text{if } GC = (GC_1 \texttt{ [] } GC_2) \end{cases}$$

Second, $\mathsf{weight}(GC)$ determines the total probability of all guards that hold:

$$\mathsf{weight}(GC) = \begin{cases} [B] \cdot E, & \text{if } GC = (B \xrightarrow{E} C) \\ \mathsf{weight}(GC_1) + \mathsf{weight}(GC_2) & \text{if } GC = (GC_1 \texttt{ [] } GC_2) \end{cases}$$

The subdistribution obtained from executing `if` GC `fi` on (s, ℓ) is then given by $[\![GC]\!](s, \ell)$ normalized by $\mathsf{weight}(GC)$. If no guard in GC holds, i.e. $\mathsf{guards}(GC)$ evaluates to 0, then the final subdistribution evaluates to 0 as well.

The semantics of the loop `do` GC `od` is defined as the limit of the distributions produced by its finite unrollings: If no loop guard holds, we terminate with probability one in the initial state (s, ℓ) and thus return the Dirac distribution $\delta_{(s,\ell)}$; otherwise, we return the distribution obtained from executing the loop body followed by the remaining loop unrollings.[10]

[9] Recall that we use a separate normalization step to account for this loss.

[10] Technically, our semantics computes the least fixed point of the loop's finite unrollings C_n, which is standard when defining program semantics, see e.g. [15].

5 From DPNs to PPL Programs

We now develop a PPL program C_{sim} that simulates the runs of a DPN N for a given scheduler and a set of goal states such that (1) every execution of C_{sim} corresponds to a run of N and vice versa, and (2) the probability distribution of C_{sim} equals the distribution of all of the net's runs that do not visit a goal state before all of their steps have been fired. We will discuss in Sect. 6 how this distribution can be used for process mining tasks beyond simulation.

We present the construction of C_{sim} step by step: we first discuss the setup and how we encode net states. In Sect. 5.2, we construct C_{sim}. Finally, Sect. 5.3 addresses why the constructed probabilistic program is correct.

5.1 Setup and Conventions

We first consider all dependencies needed for constructing C_{sim}. Throughout this section, we fix a DPN $N = \langle P, T, F, l, A, V, \Delta, \mathsf{pre}, \mathsf{post}\rangle$, an initial state (M_0, α_0), a scheduler $\mathfrak{S}$ of N, and a set of goal states $\mathcal{G}$. For simplicity, we assume that all data variables evaluate to rational numbers, i.e. $\Delta = \mathbb{Q}$, that $\mathcal{G}$ contains all deadlocked net states, and that membership in $\mathcal{G}$ for non-deadlocked states can be expressed as a Boolean formula isGoal over net states.[11]

Furthermore, we assume that $P = \{p_1, \ldots, p_{\#P}\}$, $T = \{t_1, \ldots, t_{\#T}\}$, and $V = \{v_1, v_2, \ldots, v_{\#V}\}$ for some natural numbers $\#P, \#T, \#V \in \mathbb{N}$.

We use the following program variables in our construction:

- For every place $p \in P$, $\underline{p}$ stores how many tokens are currently in p.
- For every variable $v \in V$, $\underline{v}$ stores the current value of v and $\underline{v'}$ is an internal program variable used for updating the value of $\underline{v}$ when firing a transition.

Every underlined variable, e.g. $\underline{p}$, corresponds to a concept of the net N, e.g. the tokens in the place p. Hence, it is straightforward to reconstruct a net state from a program state s. More formally, the marking $M[\![s]\!]$ encoded by s is given by $M[\![s]\!](p) = s(\underline{p})$ for all places $p \in P$. Analogously, the valuation $\alpha[\![s]\!]$ encoded by s is given by $\alpha[\![s]\!](v) = s(\underline{v})$ for all $v \in V_N$.

We write $\mathfrak{S}_T$ and $\mathfrak{S}_V$ to refer to the transition and data component obtained from evaluating the scheduler $\mathfrak{S}$ in the current program state. That is, if s is the current program state, then $(\mathfrak{S}_T, \mathfrak{S}_V) = \mathfrak{S}(M[\![s]\!], \alpha[\![s]\!])$. Notice that both $\mathfrak{S}_T$ and $\mathfrak{S}_V$ can be represented as (distribution) expressions over program variables. Similarly, we use the Boolean expression isGoal to check whether a (non-deadlocked) state is a goal state in $\mathcal{G}$.

Finally, we lift our notation $\underline{\ldots}$ to expressions over $V \cup V'$. That is, we denote by $\underline{\mathsf{pre}(t)}$ the expression $\mathsf{pre}(t)$ in which every variable $v \in V$ has been replaced by $\underline{v}$. Analogously, $\underline{\mathsf{post}(t)}$ is the expression $\mathsf{post}(t)$ in which every variable $v \in V$ has been replaced by $\underline{v}$ and every variable $v' \in V'$ has been replaced by $\underline{v'}$.

[11] Examples of $\mathcal{G}$ include (beside deadlocked states) the set of all states, where some final marking has been reached or a variable is above some threshold.

5.2 Simulating Net Runs in PPL

Intuitively, the PPL program C_{sim} simulates the runs by probabilistically selecting and firing enabled transitions in a loop until a goal state in $\mathcal{G}$ has been reached. In every loop iteration, will add exactly one step to the program log.

Figure 5 depicts how the above behaviour is implemented in the PPL program C_{sim}, where the PPL programs for setting up the initial net state (C_{init}), checking whether transition t_i is enabled ($B_{enabled}(t_i)$), and firing transition t_i ($C_{fire}(t_i)$) are discussed in the following.

```
$C_{init}$;
do      // $C_{loop}$: main loop fires enabled transitions until a goal state is reached
     $\neg\mathsf{isGoal} \;\land\; B_{enabled}(t_1) \xrightarrow{\mathfrak{S}_T(t_1)} C_{fire}(t_1)$
          $\vdots$
  [] $\neg\mathsf{isGoal} \;\land\; B_{enabled}(t_{\#T}) \xrightarrow{\mathfrak{S}_T(t_{\#T})} C_{fire}(t_{\#T})$
od
```

Fig. 5. The PPL program C_{sim} simulating the net N.

C_{init}. The program C_{init} below sets up the initial net state by assigning to variable $\underline{p_i}$ the number of tokens $M_0(p_i)$ initially in place p_i and to variable $\underline{v_j}$ the initial value $\alpha_0(v_j)$. We also initialize $\underline{v'_j}$ with $\alpha_0(v_j)$ such that primed and unprimed variables store the same values before every loop iteration.

$$\begin{aligned} C_{init}\colon \quad & \underline{p_1} \mathrel{:=} M_0(p_1);\; \ldots;\; \underline{p_{\#P}} \mathrel{:=} M_0(p_{\#P});\\ & \underline{v_1} \mathrel{:=} \alpha_0(v_1);\; \ldots;\; \underline{v_{\#V}} \mathrel{:=} \alpha_0(v_{\#V});\\ & \underline{v'_1} \mathrel{:=} \alpha_0(v_1);\; \ldots;\; \underline{v'_{\#V}} \mathrel{:=} \alpha_0(v_{\#V}); \end{aligned}$$

$B_{enabled}(t)$. The guard below checks whether transition $t \in T$ can be fired:

$$B_{enabled}(t)\colon \quad \underline{\mathsf{pre}(t)} \;\land\; \bigwedge_{p \in {}^\bullet t} \underline{p} \geq 1$$

$C_{fire}(t)$. For a transition $t \in T$, let ${}^\bullet t = \{q_1, \ldots, q_m\} \subseteq P$ and $t^\bullet = \{r_1, \ldots, r_n\} \subseteq P$. Moreover, let $V'(\mathsf{post}(t)) = \{u'_1, \ldots, u'_k\}$ be the variables that are potentially modified by firing t. Finally, we denote by *step*(t) the *message* in **Msg** representing a step (t, β) of the net, where β is given by the current values of the program's variables. Formally, *step*$(t) = (t, \beta)$, where $\beta(u) = \underline{u}$ if $u \in V(t)$ and $\beta(u) = \underline{u'}$ if $u \in V'(t)$. We then implement $C_{fire}(t)$ as follows:

```
$C_{\mathit{fire}}(t)$:  $\underline{q_1}$ := $\underline{q_1} - 1$;  ...;  $\underline{q_m}$ := $\underline{q_m} - 1$;          // remove tokens
         $\underline{r_1}$ := $\underline{r_1} + 1$;  ...;  $\underline{r_n}$ := $\underline{r_n} + 1$;          // add tokens
         $\underline{u'_1}$ := $\mathfrak{S}_V(u'_1)$;  ...;  $\underline{u'_k}$ := $\mathfrak{S}_V(u'_k)$;          // sampling
         observe $\underline{\mathsf{post}(t)}$;          // conditioning on the postcondition
         log $\mathit{step}(t)$;          // add the just performed step to the log
         $\underline{u_1}$ := $\underline{u'_1}$;  ...;  $\underline{u_k}$ := $\underline{u'_k}$          // update encoded data valuation
```

The program first updates the program variables that encode the marking based on ${}^\bullet t$ and $t^\bullet$. We then use the scheduler component $\mathfrak{S}_V$ to sample new values for all potentially modified variables. We also observe $\underline{\mathsf{post}(t)}$ to ensure that the sampled values satisfy the postcondition. After that, we add the just performed step to the program log. Finally, we update the encoded valuation α by assigning the values of primed variables to their unprimed counterparts.

5.3 Correctness

In this section, we show how the (sub)distribution produced by the PPL program C_{sim} relates to the probabilities of runs of the encoded net N.

To formalize this relationship, we call a program state s *observable* iff its primed and umprimed variables store the same values, i.e. $s(\underline{u}) = s(\underline{u'})$ for all $u \in V_N$. We denote by $s_{(M,\alpha)}$ the unique observable program state given by $s_{(M,\alpha)}(p) = M(p)$ for all $p \in P$ and $s_{(M,\alpha)}(\underline{u}) = \alpha(u)$ for all $u \in V_N$.

Our correctness theorem then intuitively states that running C_{loop} on observable states produces all legal runs of N with the same probability as N:

Theorem 1 (Correctness). *Let C_{loop} be the* PPL *program constructed for net N, goal states $\mathcal{G}$, and scheduler $\mathfrak{S}$ in Fig. 5. For all states (M, α) of N,*

$$\mathcal{S}[\![C_{loop}]\!](s_{(M,\alpha)}, \varepsilon) \;=\; \lambda(s,\sigma).\begin{cases} \mathbb{P}_{\mathfrak{S}}[M,\alpha]\,(\sigma \,|\, \Diamond\mathcal{G}), & \text{if } s = s_{(M',\alpha')} \\ & \quad \text{and } \sigma \in \mathsf{Runs}(M,\alpha,\mathcal{G}) \\ & \quad \text{and } (M,\alpha)[\sigma\rangle(M',\alpha') \\ 0, & \text{otherwise.} \end{cases}$$

For a detailed proof, we refer to [16].

In other words, for every initial net state (M, α), executing the main loop of C_{sim} on $s_{(M,\alpha)}$ produces the same distribution over runs as the encoded net N (for the same scheduler and set of goal states).

In particular, C_{init} always produces an observable program state corresponding to the initial net state (M_0, α_0), i.e. $[\![C_{init}]\!] = \lambda(s,\ell).\delta_{(s_{(M_0,\alpha_0)},\ell)}$. Hence, we have $\mathcal{S}[\![C_{sim}]\!](s,\varepsilon) = \mathcal{S}[\![C_{loop}]\!](s_{(M_0,\alpha_0)},\varepsilon)$ regardless of the initial program state s. That is, C_{sim} produces all legal runs of N starting in (M_0, α_0) with the same probability as the net for the given scheduler.

6 Probabilistic Programming for Process Mining Tasks

So far, we outlined how to construct a probabilistic program from a DPN with a scheduler $\mathfrak{S}$ that, by Theorem 1, computes every DPN run with exactly the probability induced by $\mathfrak{S}$. Our probabilistic program can be viewed both as a program that can be executed and as a statistical model that can be further analyzed with statistical inference engines. In this section, we outline how one can leverage these views for Process Mining tasks.

Log Generation (with Guarantees). By viewing probabilistic programs as executable programs, this use case immediately follows from the run (and, eventually, trace) generation capabilities of our approach. Since every execution of the probabilistic program C_{sim} yields a DPN run, it suffices to execute C_{sim} n times to generate a data set of n DPN runs, which can be further projected to obtain an event log. Notice that such projections are done at the run-to-trace level and require matching every step (t, β) to an event (as well as well-crafted handling of silent transitions, if any). Like that, each such event carries "activity payloads" with information about all the valuations of process variables from V_N involved in executing non-silent activity $l(t)$ and new valuations for the variables updated by $\mathsf{post}(t)$. By using PP, we get statistical guarantees on the data set (cf. [11]): the probability of the generated runs will converge to the run's probability induced by the selected scheduler. The same guarantees apply to the logs extracted from the sets of runs.

Distribution Analysis. By viewing probabilistic programs as statistical models, we can leverage statistical inference engines (cf. [11,12]) to analyze the distribution of DPN runs produced by our program. Knowing this distribution is useful to identify, for example, whether certain runs are particularly (un)likely. The true benefit, however, is that inference engines can also compute *conditional* probabilities, e.g. "what is the probability of reaching a marking *given* that the data variable x is at least 17.5 and that transition t_3 has been fired at most twice?". Technically, this can be achieved by inserting the command `observe` $(x > 17.5 \wedge \#t_3 \leq 2)$ in our program, where $\#t_3$ is an injected variable that counts the number of transition firings. After that, we run an inference engine to compute the conditional probability distribution over DPN runs in which the above observation holds.

The same reasoning can be adopted for generating traces with *rare events*. Assume that we know from real data that a certain event rarely happens (e.g., a transition being executed twice or two data variables being equal). We can focus on such rare events using conditional probabilities and add a suitable command `observe` φ (where φ encodes that rare event) in the probabilistic program. A statistical inference engine will then produce the conditional probability distribution over only those runs in which the rare event (defined in φ) happens, which enables further analysis, e.g. what events appear frequently if φ holds.

What-If Analysis. Along the same vein, to test a hypothesis over the given DPN model, it suffices to modify $\mathfrak{S}$ and/or add conditioning commands `observe` φ. As in the previous case, this will produce conditional probability distributions over those runs in which described scenario happens. Notice that such an analysis does not require any modifications of the underlying DPN and requires minor adjustments to the scheduler and/or observations, which can be easily incorporated without re-running the whole translation process discussed in Sect. 5.2.

7 Conclusion and Future Work

This paper, for the first time, establishes a systematic and proven-correct connection between data Petri nets (DPNs) and probabilistic programming, with the goal of making powerful simulation and inference engines available to DPNs. Such engines can be used for a plethora of tasks such as trace generation, complex statistical analysis of DPN runs, what-if analysis.

To test the feasibility of the presented approach, we developed a proof-of-concept implementation of the translation (available at https://tinyurl.com/2xx2fhvd and https://doi.org/10.5281/zenodo.12723519) along with several examples from PNML (https://www.pnml.org) representations of DPNs into WebPPL [11] programs that can subsequently be executed in the WebPPL environment, facilitating the simulation, analysis, and inference of the statistical model they represent.

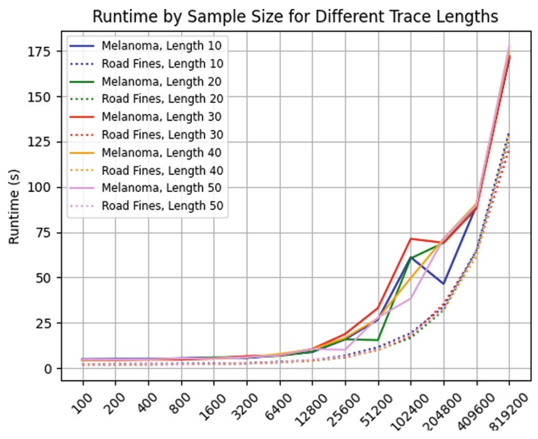

Fig. 6. Evaluation results for Melanoma and Road Fine, highlighting the runtime (in seconds) in relation to the number of generated runs (x-axis)

The simulator was tested on two nets: the Road Fine DPN taken from [18] (9 places, 19 transitions, 11 guards, 8 variables) and the Melanoma DPN (50 places, 76 transitions, 52 guards, 26 variables) taken from [13]. For both DPNs, we used a scheduler that uniformly selects transitions and data values. The set of goal states consists of all states that are reached by runs of some fixed length. WebPPL's MCMC inference engine was executed for various run lengths (10–50) and sample sizes (100-819200) with a 180 s timeout to evaluate the performance. Figure 6 illustrates the runtime across five computation cycles. Notice that the number of generated runs, depicted on the x-axis, is doubled in every step, hence the exponential increase in runtime. Since our implementation naively follows the formal translation in Sect. 5, we consider the obtained runtimes encouraging and believe that there is ample space for optimizations.

For future work, we would like to investigate how probabilistic programming can be used to handle different types of monitoring tasks, where a partial execution prefix is given and has to be reproduced by the probabilistic program. Moreover, our current work does not explicitly support silent transitions. Finally, we want to study extended simulation setups where dependencies between different runs or non-functional criteria (e.g., case arrival time or resource allocation [24]) are also taken into account.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Burattin, A.: PLG2: multiperspective processes randomization and simulation for online and offline settings. arXiv, abs/1506.08415 (2015)
3. Chick, S.E.: Bayesian ideas and discrete event simulation: why, what and how. In: Perrone, L.F., Lawson, B., Liu, J., Wieland, F.P. (eds.) Proceedings of the WSC, pp. 96–105. IEEE Computer Society (2006)
4. Dahlqvist, F., Kozen, D.: Semantics of higher-order probabilistic programs with conditioning. Proc. ACM Program. Lang. **4**(POPL), 57:1–57:29 (2020)
5. de Leoni, M., Felli, P., Montali, M.: A holistic approach for soundness verification of decision-aware process models. In: Trujillo, J.C., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 219–235. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_17
6. de Leoni, M., Felli, P., Montali, M.: Integrating BPMN and DMN: modeling and analysis. J. Data Semant. **10**(1–2), 165–188 (2021)
7. Felli, P., Gianola, A., Montali, M., Rivkin, A., Winkler, S.: CoCoMoT: conformance checking of multi-perspective processes via SMT. In: Polyvyanyy, A., Wynn, M.T., Van Looy, A., Reichert, M. (eds.) BPM 2021. LNCS, vol. 12875, pp. 217–234. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85469-0_15
8. Felli, P., Gianola, A., Montali, M., Rivkin, A., Winkler, S.: Conformance checking with uncertainty via SMT. In: Di Ciccio, C., Dijkman, R., del Río Ortega, A., Rinderle-Ma, S. (eds.) BPM 2022. LNCS, vol. 13420, pp. 199–216. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16103-2_15
9. Felli, P., Montali, M., Winkler, S.: CTL* model checking for data-aware dynamic systems with arithmetic. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 36–56. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_4
10. Felli, P., Montali, M., Winkler, S.: Soundness of data-aware processes with arithmetic conditions. In: Franch, X., Poels, G., Gailly, F., Snoeck, M. (eds.) CAiSE 2022. LNCS, vol. 13295, pp. 389–406. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07472-1_23
11. Goodman, N.D., Stuhlmüller, A.: The Design and Implementation of Probabilistic Programming Languages (2014). http://dippl.org. Accessed 8 Mar 2024
12. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
13. Grüger, J., Geyer, T., Kuhn, M., Braun, S.A., Bergmann, R.: Verifying guideline compliance in clinical treatment using multi-perspective conformance checking: a case study. In: Munoz-Gama, J., Lu, X. (eds.) ICPM 2021. LNBIP, vol. 433, pp. 301–313. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-98581-3_22

14. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Cham (2009)
15. Kozen, D.: Semantics of probabilistic programs. In: FOCS. IEEE (1979)
16. Kuhn, M., Grüger, J., Matheja, C., Rivkin, A.: Data Petri nets meet probabilistic programming (extended version). arXiv (2024)
17. Law, A.M.: Simulation Modeling & Analysis. McGraw-Hill, New York (2015)
18. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing **98**(4) (2016)
19. Mannhardt, F., Leemans, S.J.J., Schwanen, C.T., de Leoni, M.: Modelling data-aware stochastic processes - discovery and conformance checking. In: Gomes, L., Lorenz, R. (eds.) PETRI NETS 2023. LNCS, vol. 13929, pp. 77–98. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33620-1_5
20. Medeiros, A., Günther, C.: Process mining: using CPN tools to create test logs for mining algorithms. In: CPN (2004)
21. Mitsyuk, A.A., Shugurov, I.S., Kalenkova, A.A., van der Aalst, W.M.: Generating event logs for high-level process models. Simul. Model. Pract. Theory **74** (2017)
22. Pufahl, L., Wong, T.Y., Weske, M.: Design of an extensible BPMN process simulator. In: Teniente, E., Weidlich, M. (eds.) BPM 2017. LNBIP, vol. 308, pp. 782–795. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74030-0_62
23. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming, 1st edn. Wiley, Hoboken (1994)
24. Rosenthal, K., Ternes, B., Strecker, S.: Business process simulation on procedural graphical process models. Bus. Inf. Syst. Eng. **63**(5), 569–602 (2021)
25. van de Meent, J.-W., Paige, B., Yang, H., Wood, F.: An introduction to probabilistic programming. arXiv preprint arXiv:1809.10756 (2018)
26. Aalst, W.M.P.: Business process simulation survival guide. In: vom Brocke, J., Rosemann, M. (eds.) Handbook on Business Process Management 1. IHIS, pp. 337–370. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-642-45100-3_15
27. van der Aalst, W.M.P.: Process mining and simulation: a match made in heaven! In: SummerSim, pp. 4:1–4:12. ACM (2018)