Bisimulation and simulation algorithms on probabilistic transition systems by abstract interpretation

Silvia Crafa · Francesco Ranzato

Abstract We show how bisimulation equivalence and simulation preorder on probabilistic LTSs (PLTSs), namely the main behavioural relations on probabilistic nondeterministic processes, can be characterized by abstract interpretation. Both bisimulation and simulation can be obtained as completions of partitions and preorders, viewed as abstract domains, w.r.t. a pair of concrete functions that encode a PLTS. This approach provides a general framework for designing algorithms that compute bisimulation and simulation on PLTSs. Notably, (i) we show that the standard bisimulation algorithm by Baier et al. (2000) can be viewed as an instance of such a framework and (ii) we design a new efficient simulation algorithm that improves the state of the art.

Keywords Probabilistic LTS · Probabilistic bisimulation equivalence · Probabilistic simulation preorder · Abstract interpretation · Simulation algorithm

1 Introduction

Randomization phenomena in concurrent systems have been widely studied in probabilistic extensions of process algebras like Markov chains and probabilistic labeled transition systems (PLTSs). Most standard tools for studying nonprobabilistic processes, like behavioural equivalences, temporal logics and model checking, have been investigated for these probabilistic models. In particular, bisimulation equivalence and simulation preorder relations, namely the main behavioural relations between concurrent systems, have been extended and studied in a probabilistic setting [6, 10, 16, 17].

Abstract interpretation [2, 3] is a well-known general theory for specifying the approximation of formal semantics. Abstract domains play an essential role in any abstract interpretation design, since they encode in an ordered structure how concrete semantic properties are approximated. A number of behavioural relations, including bisimulation, stuttering bisimulation and simulation, have been characterized in abstract interpretation as complete

S. Crafa · F. Ranzato (✉)
Dipartimento di Matematica, University of Padova, Padova, Italy
e-mail: ranzato@math.unipd.it

refinements, so-called forward complete shells, of abstract domains w.r.t. logical/temporal operators of suitable modal logics [14]. One notable benefit of this approach is that it provides a general framework for designing basic algorithms that compute behavioral relations as forward complete shells of abstract domains. As a remarkable example, this abstract interpretation-based approach led to an efficient algorithm for computing the simulation preorder for nonprobabilistic processes [13, 15] that features the best time complexity among the simulation algorithms.

In this paper we show how the abstract interpretation approach can be applied to probabilistic LTSs in order (i) to characterize bisimulation equivalence and simulation preorder as logical completions of abstract domains and (ii) to design bisimulation and simulation algorithms.

Main results We consider probabilistic processes specified as PLTSs, a general model that exhibits both non-deterministic choice (as in LTSs) and probabilistic choice (as in Markov chains). In [14], bisimulation in LTSs has been characterized in terms of forward complete shells of partitions w.r.t. the predecessor operator of LTSs. We show that this same idea scales to the case of PLTSs by considering the probabilistic predecessor operator that defines the transitions of a PLTS together with a probabilistic function that encodes the distributions in the PLTS (this latter operator is somehow reminiscent of a probabilistic connective in Parma and Segala's [12] modal logics for probabilistic bisimulation and simulation). Bisimulation equivalence in PLTSs is thus characterized as a domain refinement through a complete shell w.r.t. the above two operators. On the other hand, the simulation preorder in PLTSs turns out to be the same complete shell of abstract domains w.r.t. the same two operators, but using different underlying abstract domains: for bisimulation, the complete shell is computed in a space of abstractions that are state and distribution partitions, while for simulation the same complete shell is instead computed over abstractions that are preorders on states and distributions.

Complete shells of abstract domains may in general be obtained through a simple fixpoint computation. We show how such a basic procedure can be instantiated to obtain two algorithms that iteratively compute bisimulation and simulation on PLTSs. Interestingly, the standard procedure for computing bisimulations in PLTSs, namely Baier-Engelen-Majster's algorithm [1], can be actually viewed as an implementation of our complete shell procedure that characterizes bisimulation. On the other hand, we show that the corresponding complete shell for computing the simulation preorder yields a new efficient probabilistic simulation algorithm that advances the state of the art: in fact, its time and space complexity bounds improve on the best known simulation algorithm for PLTSs by Zhang et al. [20].

This is an extended and revised version of the conference paper [4].

2 Bisimulation and simulation in PLTSs

Given a set X, $\text{Distr}(X)$ denotes the set of (stochastic) distributions on X, i.e., functions $d : X \to [0, 1]$ such that $\sum_{x \in X} d(x) = 1$. The support of a distribution d is defined by $\text{supp}(d) \triangleq \{x \in X \mid d(x) > 0\}$; also, if $S \subseteq X$ then $d(S) \triangleq \sum_{s \in S} d(s)$. The Dirac distribution on $x \in X$, denoted by δ_x, is the distribution that assigns probability 1 to x (and 0 otherwise).

A probabilistic LTS (PLTS) is a tuple $\mathcal{S} = \langle \Sigma, Act, \to \rangle$ where Σ is a set of states, Act is a set of actions and $\to \subseteq \Sigma \times Act \times \text{Distr}(\Sigma)$ is a transition relation, where $(s, a, d) \in \to$ is also denoted by $s \xrightarrow{a} d$. We denote by $\text{Distr} \triangleq \{d \in \text{Distr}(\Sigma) \mid \exists s \in \Sigma. \exists a \in Act . s \xrightarrow{a} d\}$

the set of target distributions in $\mathcal{S}$. Given $D \subseteq \text{Distr}$, we write $s \xrightarrow{a} D$ when there exists $d \in D$ such that $s \xrightarrow{a} d$. For any $a \in Act$, the predecessor and successor operators $\text{pre}_a : \wp(\text{Distr}) \to \wp(\Sigma)$ and $\text{post}_a : \wp(\Sigma) \to \wp(\text{Distr})$ are defined as usual by $\text{pre}_a(D) \triangleq \{s \in \Sigma \mid s \xrightarrow{a} D\}$ and $\text{post}_a(S) \triangleq \{d \in \text{Distr} \mid \exists s \in S. s \xrightarrow{a} d\}$. For any $d \in \text{Distr}$ and $s \in \Sigma$, we define $\text{in}(d) \triangleq \{a \in Act \mid \text{pre}_a(\{d\}) \neq \varnothing\}$ and $\text{out}(s) \triangleq \{a \in Act \mid \text{post}_a(\{s\}) \neq \varnothing\}$.

Bisimulation A partition of a set X is a set $P \subseteq \wp(X)$ of nonempty subsets of X (called blocks) that are disjoint and whose union gives X. Let $\text{Part}(X)$ denote the set of partitions of X. If $P \in \text{Part}(X)$ and $x \in X$ then $P(x)$ denotes the unique block of P that contains x. A partition P will be also viewed as a mapping $P : \wp(X) \to \wp(X)$ defined by $P(Y) \triangleq \bigcup_{y \in Y} P(y)$. Any partition $P \in \text{Part}(X)$ induces an equivalence relation (which can be equivalently given as a partition) over distributions $\equiv_P \in \text{Part}(\text{Distr}(X))$ which is defined as follows: for any $d, e \in \text{Distr}(X)$, $d \equiv_P e$ if for any $B \in P$, $d(B) = e(B)$. In words, two distributions are $\equiv_P$-equivalent whenever they give the same probability to the blocks of P.

Given a PLTS $\mathcal{S} = \langle \Sigma, Act, \to \rangle$, a partition $P \in \text{Part}(\Sigma)$ is a bisimulation on $\mathcal{S}$ when for all $s, t \in \Sigma$ and $d \in \text{Distr}$, if $P(s) = P(t)$ and $s \xrightarrow{a} d$ then there exists $e \in \text{Distr}$ such that $t \xrightarrow{a} e$ and $d \equiv_P e$. Bisimilarity $P_{\text{bis}} \in \text{Part}(\Sigma)$ is defined as follows: for any $s \in \Sigma$, $P_{\text{bis}}(s) \triangleq \bigcup\{P(s) \mid P \text{ is a bisimulation on } \mathcal{S}\}$. P_{bis} turns out to be the greatest bisimulation on $\mathcal{S}$ which is also called the bisimulation partition on $\mathcal{S}$.

Simulation A preorder on a set X is a reflexive and transitive relation $R \subseteq X \times X$. Let $\text{PreOrd}(X)$ denote the set of preorders on X. If $R \in \text{PreOrd}(X)$ and $S \subseteq X$ then $R(S) \triangleq \{x \in X \mid \exists s \in S. (s, x) \in R\}$ denotes the image of S for R. Similarly to the case of partitions, any preorder $R \in \text{PreOrd}(X)$ induces a preorder $\leq_R$ on $\text{Distr}(X)$ which is defined as follows: for any $d, e \in \text{Distr}(X)$, $d \leq_R e$ if for any $S \subseteq X$, $d(S) \leq e(R(S))$; we will sometimes use the equivalent definition $d(R(S)) \leq e(R(S))$ ([19]). Such a definition of $\leq_R$ can be equivalently stated in terms of so-called weight functions between distributions and of maximum flows between networks. We briefly recall its equivalent formulation based on the maximum flow problem since our simulation algorithm, as well as the simulation algorithms by Baier et al. [1] and Zhang et al. [20], are based on this notion.

Let $\overline{X} \triangleq \{\overline{x} \mid x \in X\}$, where $\overline{x}$ are pairwise distinct new elements; $\bot$ (the source) and $\top$ (the sink) are a pair of new distinct elements not contained in $X \cup \overline{X}$. Let $R \subseteq X \times X$ and $d, e \in \text{Distr}(X)$. The network $\mathcal{N}(d, e, R)$ is defined as follows: the set of nodes is $V \triangleq \text{supp}(d) \cup \overline{\text{supp}(e)} \cup \{\bot, \top\}$ while the set of edges $E \subseteq V \times V$ is defined by

$$E \triangleq \big\{(x, \overline{y}) \mid (x, y) \in R\big\} \cup \big\{(\bot, x) \mid x \in \text{supp}(d)\big\} \cup \big\{(\overline{y}, \top) \mid y \in \text{supp}(e)\big\}.$$

The capacity function $c : V \times V \to [0, 1]$ is defined as follows: for all $x \in \text{supp}(d)$, $c(\bot, x) \triangleq d(x)$; for all $y \in \text{supp}(e)$, $c(\overline{y}, \top) \triangleq e(y)$; for all the remaining edges $(x, \overline{y}) \in E$, $c(x, \overline{y}) \triangleq 1$. It turns out that $d \leq_R e$ if and only if the maximum flow of the network $\mathcal{N}(d, e, R)$ is 1 (see [1, 5, 19, 20]).

Given a PLTS $\mathcal{S} = \langle \Sigma, Act, \to \rangle$, a preorder $R \in \text{PreOrd}(\Sigma)$ is a simulation on $\mathcal{S}$ when for all $s, t \in \Sigma$ and $d \in \text{Distr}$, if $t \in R(s)$ and $s \xrightarrow{a} d$ then there exists $e \in \text{Distr}$ such that $t \xrightarrow{a} e$ and $d \leq_R e$. The simulation preorder $R_{\text{sim}} \in \text{PreOrd}(\Sigma)$ on $\mathcal{S}$ is defined as follows: for all $s \in \Sigma$, $R_{\text{sim}}(s) \triangleq \bigcup\{R(s) \mid R \text{ is a simulation on } \mathcal{S}\}$. It turns out that R_{sim} is the greatest simulation preorder on $\mathcal{S}$. The simulation partition P_{sim} on $\mathcal{S}$ is the kernel of the simulation preorder, i.e., for all $s, t \in \Sigma$, $P_{\text{sim}}(s) = P_{\text{sim}}(t)$ iff $s \in R_{\text{sim}}(t)$ and $t \in R_{\text{sim}}(s)$.

Example 2.1 Consider the PLTS depicted in Fig. 1, where $\Sigma = \{s_1, s_2, s_3, x_1, \ldots, x_6, t, u, v, w\}$, $Act = \{a, b, c, d\}$ and $\text{Distr} = \{d_1, d_2, d_3, \delta_t, \delta_u, \delta_v, \delta_w\}$. We have that s_2 simulates

Fig. 1 A PLTS
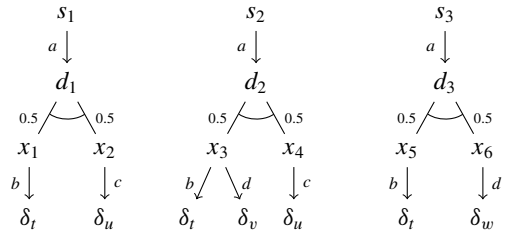
s_1 while s_1 does not simulate s_2 since starting from s_2 a d-transition can be fired, whereas starting from s_1 this is not possible. Moreover, even if x_3 simulates both x_5 and x_6, we have that $d_3 \not\leq_{R_{\text{sim}}} d_2$ since $1 = d_3(\{x_5, x_6\}) \not\leq d_2(R_{\text{sim}}(\{x_5, x_6\})) = 0.5$ since $x_4 \notin R_{\text{sim}}(\{x_5, x_6\})$. Therefore, s_2 does not simulate s_3.

3 Shells

3.1 Forward completeness

In standard abstract interpretation [2, 3], approximations of a concrete semantic domain are encoded by abstract domains (or abstractions), that are specified by Galois insertions (GIs for short) or, equivalently, by adjunctions. Concrete and abstract domains are defined as complete lattices $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$ where $x \leq y$ means that y approximates x both concretely and abstractly. A GI of A into C is determined by a surjective abstraction map $\alpha : C \to A$ and an injective concretization map $\gamma : A \to C$ such that $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$, and is denoted by (α, C, A, γ).

Lattices of abstract domains Recall that GIs of a common concrete domain C are pre-ordered w.r.t. their relative precision: $\mathcal{G}_1 = (\alpha_1, C, A_1, \gamma_1) \trianglelefteq \mathcal{G}_2 = (\alpha_2, C, A_2, \gamma_2)$, i.e. A_1 is a refinement of A_2 or, equivalently, A_2 is a simplification of A_1, if and only if for all $c \in C$, $\gamma_1(\alpha_1(c)) \leq_C \gamma_2(\alpha_2(c))$. Moreover, $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent when $\mathcal{G}_1 \trianglelefteq \mathcal{G}_2$ and $\mathcal{G}_2 \trianglelefteq \mathcal{G}_1$. We denote by $\text{Abs}(C)$ the family of abstract domains of C up to the above equivalence. It is well known ([3]) that $\langle \text{Abs}(C), \trianglelefteq \rangle$ is a complete lattice. Given a family of abstract domains $\mathcal{X} \subseteq \text{Abs}(C)$, their lub $\bigsqcup \mathcal{X}$ is therefore the most precise domain in $\text{Abs}(C)$ which is a simplification of any domain in $\mathcal{X}$.

Forward complete abstractions Let $f : C \to D$ be some concrete semantic function defined on a pair of concrete domains C and D, and let $A \in \text{Abs}(C)$ and $B \in \text{Abs}(D)$ be a pair of abstractions. In the following, we will denote by $\sqsubseteq_{X \to Y}$ the pointwise ordering relation between functions in $X \to Y$. Given an abstract function $f^\sharp : A \to B$, we have that $\langle A, B, f^\sharp \rangle$ is a sound abstract interpretation of f when $f \circ \gamma_{A,C} \sqsubseteq_{A \to D} \gamma_{B,D} \circ f^\sharp$. Forward completeness [3, 7] corresponds to the situation where the following diagram commutes:

$$
\begin{array}{ccc}
C & \xrightarrow{f} & D \\
\gamma_{A,C} \uparrow & & \uparrow \gamma_{B,D} \\
A & \xrightarrow{f^\sharp} & B
\end{array}
$$

that is $f \circ \gamma_{A,C} = \gamma_{B,D} \circ f^\sharp$, meaning that the abstract function $f^\sharp$ is able to replicate the behaviour of f on the abstract domains A and B with no loss of precision. If an abstract interpretation $\langle A, B, f^\sharp \rangle$ is forward complete then it turns out that the abstract function $f^\sharp$

indeed coincides with $\alpha_{D,B} \circ f \circ \gamma_{A,C}$, which is the best correct approximation of the concrete function f on the pair of abstractions $\langle A, B \rangle$. Hence, the notion of forward completeness of an abstract interpretation $\langle A, B, f^{\sharp} \rangle$ does not depend on the choice of the abstract function $f^{\sharp}$ but only depends on the chosen abstract domains A and B. Accordingly, a pair of abstract domains $\langle A, B \rangle \in \text{Abs}(C) \times \text{Abs}(D)$ is called forward complete for f (or simply f-complete) iff $f \circ \gamma_{A,C} = \gamma_{B,D} \circ (\alpha_{D,B} \circ f \circ \gamma_{A,C})$. It is not difficult to prove that this is equivalent to $\{f(\gamma_{A,C}(a)) \mid a \in A\} \subseteq \{\gamma_{B,D}(b) \mid b \in B\}$, i.e., $f(\gamma_{A,C}(A)) \subseteq \gamma_{B,D}(B)$. Hence the pair $\langle A, B \rangle$ is f-complete iff the image of f in D is contained in $\gamma_{B,D}(B)$. If $\mathcal{F} \subseteq C \to D$ is a set of concrete functions then $\langle A, B \rangle$ is $\mathcal{F}$-complete when $\langle A, B \rangle$ is f-complete for all $f \in \mathcal{F}$.

3.2 Shells of abstract domains

Given a set of semantic functions $\mathcal{F} \subseteq C \to D$ and a pair of abstractions $\langle A, B \rangle \in \text{Abs}(C) \times \text{Abs}(D)$, the notion of forward complete shell [7] formalizes the problem of finding the most abstract pair $\langle A', B' \rangle$ such that $A' \trianglelefteq A$, $B' \trianglelefteq B$ and $\langle A', B' \rangle$ is $\mathcal{F}$-complete, which is a particular case of abstraction refinement [8]. It turns out (see [7]) that any pair $\langle A, B \rangle$ can be minimally refined to its forward $\mathcal{F}$-complete shell:

$$\text{Shell}_{\mathcal{F}}(\langle A, B \rangle)$$
$$\triangleq \bigsqcup \{ \langle A', B' \rangle \in \text{Abs}(C) \times \text{Abs}(D) \mid \langle A', B' \rangle \trianglelefteq \langle A, B \rangle, \langle A', B' \rangle \text{ is } \mathcal{F}\text{-complete} \}.$$

Thus, $\text{Shell}_{\mathcal{F}}(\langle A, B \rangle)$ encodes the least refinement of a pair of abstractions $\langle A, B \rangle$ which is needed in order to achieve forward completeness for $\mathcal{F}$.

Let us now consider a further set of concrete semantic functions $\mathcal{G} \subseteq D \to C$ that operate in the opposite direction w.r.t. $\mathcal{F}$, i.e., from D to C. Given $A \in \text{Abs}(C)$ and $B \in \text{Abs}(D)$, it makes sense to consider both forward $\mathcal{F}$-completeness of $\langle A, B \rangle$ and forward $\mathcal{G}$-completeness of the reversed pair $\langle B, A \rangle$. Thus, $\langle A, B \rangle$ is defined to be $\langle \mathcal{F}, \mathcal{G} \rangle$-complete when $\langle A, B \rangle$ is $\mathcal{F}$-complete and $\langle B, A \rangle$ is $\mathcal{G}$-complete. Here again, any pair $\langle A, B \rangle$ can be minimally refined to its $\langle \mathcal{F}, \mathcal{G} \rangle$-complete shell:

$$\text{Shell}_{\langle \mathcal{F}, \mathcal{G} \rangle}(\langle A, B \rangle)$$
$$\triangleq \bigsqcup \{ \langle A', B' \rangle \in \text{Abs}(C) \times \text{Abs}(D) \mid \langle A', B' \rangle \trianglelefteq \langle A, B \rangle, \langle A', B' \rangle \text{ is } \langle \mathcal{F}, \mathcal{G} \rangle\text{-complete} \}.$$

Such a combined shell $\text{Shell}_{\langle \mathcal{F}, \mathcal{G} \rangle}(\langle A, B \rangle)$ can be obtained through the ShellAlgo() procedure described in Fig. 2. This procedure works by iteratively refining the abstractions A and B separately until both $\langle A, B \rangle$ becomes $\mathcal{F}$-complete and $\langle B, A \rangle$ becomes $\mathcal{G}$-complete. The ShellAlgo() procedure crucially relies on the Stabilize() function. In general, given a set of functions $\mathcal{H}$ and a pair of abstractions $\langle X, Y \rangle$, we have that Stabilize($\mathcal{H}, X, Y$) refines the abstraction Y to $Y_{\text{stable}} \triangleq \bigsqcup \{ Y' \mid Y' \trianglelefteq Y, \langle X, Y' \rangle \text{ is } \mathcal{H}\text{-complete} \}$, so that $\langle X, Y_{\text{stable}} \rangle$ becomes $\mathcal{H}$-stable (i.e., $\mathcal{H}$-complete). Notice that the parameter Y is an input-output parameter. Hence, Stabilize($\mathcal{F}, A, B$) minimally refines B to B' so that $\langle A, B' \rangle$ is $\mathcal{F}$-complete. Hence, while the abstraction B is refined, the abstraction A is left unchanged. Moreover, if B is actually refined into $B' \lhd B$, then the $\mathcal{G}$-Stable flag is set to false so that ShellAlgo() proceeds by $\mathcal{G}$-stabilizing $\langle B', A \rangle$, i.e., by calling Stabilize($\mathcal{G}, B', A$). Thus, ShellAlgo($\mathcal{F}, \mathcal{G}, A, B$) begins by first checking whether $\langle A, B \rangle$ is $\mathcal{F}$-complete and $\langle B, A \rangle$ is $\mathcal{G}$-complete, and then proceeds by iteratively refining the abstractions A and B separately, namely it refines B w.r.t. $\mathcal{F}$ while A is kept fixed and then it refines A w.r.t. $\mathcal{G}$ while B is kept fixed. It turns out

```
1 ShellAlgo(𝓕, 𝓖, A, B) {
2    Initialize();
3    while ¬(𝓕-Stable ∧ 𝓖-Stable) do
4       if ¬𝓕-Stable then 𝓖-Stable := Stabilize(𝓕, A, B); 𝓕-Stable := true;
5       if ¬𝓖-Stable then 𝓕-Stable := Stabilize(𝓖, B, A); 𝓖-Stable := true;
6    return ⟨A, B⟩;
7 }
```

```
1 Initialize() {
2    𝓕-Stable := CheckStability(𝓕, A, B); 𝓖-Stable := CheckStability(𝓖, B, A);
3 }
```

```
1 bool Stabilize(𝓗, X, Y) {
2    Y_old := Y;
3    Y := ⊔{Y′ ∈ Abs | Y′ ⊴ Y, ⟨X, Y′⟩ is 𝓗-complete};
4    return (Y = Y_old);
5 }
```

Fig. 2 Basic shell algorithm

that the ShellAlgo() procedure is a nontrivial gneralization of a basic forward completion procedure [7], that corresponds to the case where $A = B, C = D, 𝓕 = 𝓖$.

Theorem 3.1 ShellAlgo($𝓕, 𝓖, A, B$) = Shell$_{⟨𝓕,𝓖⟩}(⟨A, B⟩)$.

Proof Firstly, we observe in general that if $Y_{\text{stable}} ≜ ⊔\{Y′ ∈ \text{Abs} \mid Y′ ⊴ Y, ⟨X, Y′⟩$ is $𝓗$-complete} then $⟨X, Y_{\text{stable}}⟩$ is forward $𝓗$-complete. In fact, let $⟨X, Y_i⟩_{i∈I}$ be a family of $𝓗$-complete pairs with $Y_i ⊴ Y$, then by definition of forward $𝓗$-completeness we have that for all $i ∈ I$ and for all $h ∈ 𝓗$, $h(γ(X)) ⊆ γ(Y_i)$, hence $h(γ(X)) ⊆ ⋂_{i∈I} γ(Y_i)$, that is $h(γ(X)) ⊆ γ(⊔_{i∈I} Y_i)$; therefore, $⟨X, ⊔_{i∈I} Y_i⟩$ is $𝓗$-complete. Secondly, the procedure ShellAlgo() always terminates because if $⟨A, B⟩$ and $⟨A′, B′⟩$ are, respectively, the current abstraction pairs at the beginning and at the end of a single iteration of the while-loop, then either $A′ ⊲ A$ or $B′ ⊲ B$. Let $⟨A_{\text{ref}}, B_{\text{ref}}⟩$ be the output abstraction pair of ShellAlgo($𝓕, 𝓖, A, B$) and let $⟨A^s, B^s⟩ ≜ $ Shell$_{⟨𝓕,𝓖⟩}(⟨A, B⟩)$. Then, by the observation above, $⟨A_{\text{ref}}, B_{\text{ref}}⟩$ is $𝓕$-complete and $⟨B_{\text{ref}}, A_{\text{ref}}⟩$ is $𝓖$-complete, so that $⟨A_{\text{ref}}, B_{\text{ref}}⟩ ⊴ ⟨A^s, B^s⟩$. Conversely, since $⟨A^s, B^s⟩$ is $𝓕$-complete and $⟨B^s, A^s⟩$ is $𝓖$-complete then we have that ShellAlgo($𝓕, 𝓖, A^s, B^s$) = $⟨A^s, B^s⟩$. Also, note that ShellAlgo() is monotone, namely, if $⟨A_1, B_1⟩ ⊴ ⟨A_2, B_2⟩$ then ShellAlgo($𝓕, 𝓖, A_1, B_1$) ⊴ ShellAlgo($𝓕, 𝓖, A_2, B_2$). Thus, from $⟨A^s, B^s⟩ ⊴ ⟨A, B⟩$, we derive that $⟨A^s, B^s⟩ = $ ShellAlgo($𝓕, 𝓖, A^s, B^s$) ⊴ ShellAlgo($𝓕, 𝓖, A, B$) = $⟨A_{\text{ref}}, B_{\text{ref}}⟩$. ☐

4 Bisimulation as a shell

Bisimulation is commonly computed by coarsest partition refinement algorithms [1, 11] that iteratively refine a current partition until it becomes the bisimulation partition. Coarsest partition refinements can be formalized as shells of partitions: given a property of partitions, i.e., any subset of partitions $ℙ ⊆ \text{Part}(X)$, the $ℙ$-shell of $Q ∈ \text{Part}(X)$ corresponds to the coarsest partition refinement of Q that satisfies $ℙ$, when this exists. In this section we show how bisimulation in PLTSs can be equivalently stated in terms of forward complete shells of partitions w.r.t. suitable concrete semantic functions. We also show how the above basic shell algorithm ShellAlgo() can be instantiated to compute bisimulations on PLTSs.

4.1 Shells of partitions

Let us first recall that, given a finite set X, $\langle \text{Part}(X), \preceq, \curlyvee, \curlywedge \rangle$ is a (finite) lattice where $P_1 \preceq P_2$ (i.e., P_2 is coarser than P_1 or P_1 refines P_2) iff $\forall x \in X. P_1(x) \subseteq P_2(x)$, and its top element is $\top_{\text{Part}(X)} = \{X\}$. By following the approach in [14], any partition $P \in \text{Part}(X)$ can be viewed as an abstraction of $\langle \wp(X), \subseteq \rangle$, where any set $S \subseteq X$ is approximated through its minimal cover in the partition P. This is formalized by viewing P as the abstract domain $\text{closed}(P) \triangleq \{S \subseteq X \mid P(S) = S\}$, remind that $P(S) \triangleq \bigcup_{s \in S} P(s)$. In other terms, $S \in \text{closed}(P)$ iff $S = \bigcup_{i \in I} B_i$ for some blocks $\{B_i\}_{i \in I} \subseteq P$. Note that $\varnothing, X \in \text{closed}(P)$ and that $\langle \text{closed}(P), \subseteq, \cup, \cap \rangle$ is a lattice. It turns out that $\langle \text{closed}(P), \subseteq \rangle$ is an abstraction in $\langle \text{Abs}(\wp(X)), \subseteq \rangle$, where any set $S \subseteq X$ is approximated through the blocks in P covering S, namely by $\bigcup\{B \in P \mid B \cap S \neq \varnothing\} \in \text{closed}(P)$.

The above embedding of partitions as abstract domains allows us to define a notion of forward completeness for partitions. Let $f : \wp(X) \to \wp(Y)$ be a concrete semantic function that transforms subsets. Then, a pair of partitions $\langle P, Q \rangle \in \text{Part}(X) \times \text{Part}(Y)$ is (forward) f-complete when the pair of abstract domains $\langle \text{closed}(P), \text{closed}(Q) \rangle$ is forward complete for f, that is, $f(\text{closed}(P)) \subseteq \text{closed}(Q)$. In other terms, since $U \in \text{closed}(P)$ iff U is a union of blocks of P, i.e., $U = P(U)$, we have that $\langle P, Q \rangle$ is f-complete when for any union U of blocks of P, $f(U)$ still is a union of blocks of Q. Accordingly, we will often use the following equivalent definition.

Definition 4.1 (Forward completeness for partitions) The pair $\langle P, Q \rangle$ is f-complete when for any $S \subseteq X$, $y \in f(P(S))$ and $y' \in Q(y)$, it holds $y' \in f(P(S))$.

If we consider an additional function $g : \wp(Y) \to \wp(X)$ then $\langle P, Q \rangle$ is $\langle f, g \rangle$-complete when $\langle P, Q \rangle$ is f-complete and $\langle Q, P \rangle$ is g-complete. Analogously to forward complete shells of generic abstract domains in Sect. 3, it is easy to see that forward complete shells of partitions exist. Given $\mathcal{F} \subseteq \wp(X) \to \wp(Y)$ and $\mathcal{G} \subseteq \wp(Y) \to \wp(X)$, $\text{Shell}_{\langle \mathcal{F}, \mathcal{G} \rangle}(\langle P, Q \rangle)$ is the coarsest pair of partitions that (component-wise) refines the pair $\langle P, Q \rangle$ and is $\langle \mathcal{F}, \mathcal{G} \rangle$-complete, namely

$$
\begin{aligned}
&\text{Shell}_{\langle \mathcal{F}, \mathcal{G} \rangle}(\langle P, Q \rangle) \\
&\quad \triangleq \curlyvee \{\langle P', Q' \rangle \in \text{Part}(X) \times \text{Part}(Y) \mid \langle P', Q' \rangle \preceq \langle P, Q \rangle, \ \langle P', Q' \rangle \text{ is } \langle \mathcal{F}, \mathcal{G} \rangle\text{-complete}\}.
\end{aligned}
$$

4.2 Bisimulation on PLTSs

Ranzato and Tapparo [14] have shown that bisimulation on a LTS $\mathcal{L}$ can be equivalently defined in terms of forward complete shells of partitions w.r.t. the predecessor operator of $\mathcal{L}$. This same idea scales to the case of PLTSs taking into account that: (i) in a PLTS the target of the transition relation is a set of distributions rather than a set of states, and (ii) bisimulation on the set of states of a PLTS induces an equivalence over distributions that depends on the probabilities that distributions assign to blocks of bisimilar states. Let $\mathcal{S} = \langle \Sigma, Act, \to \rangle$ be a PLTS and consider the following two functions, where $a \in Act$ and $p \in [0, 1]$:

$$
\begin{aligned}
\text{pre}_a &: \wp(\text{Distr}) \to \wp(\Sigma), \ \text{pre}_a(D) \triangleq \{s \in \Sigma \mid s \xrightarrow{a} D\} \\
\text{prob}_p &: \wp(\Sigma) \to \wp(\text{Distr}), \ \text{prob}_p(S) \triangleq \{d \in \text{Distr} \mid d(S) \geq p\}
\end{aligned}
$$

$\text{pre}_a(D)$ is the a-predecessor function in the PLTS $\mathcal{S}$ for a set of distributions D while $\text{prob}_p(S)$ returns the distributions whose probability on the set S is greater than or equal

to p. Notice that, rather than considering any $p \in [0, 1]$, we can restrict to the weights that occur in the PLTS $\mathcal{S}$, that is the subset $\mathcal{W}_\mathcal{S} \triangleq \{p \in [0, 1] \mid \exists S \subseteq \Sigma. \exists d \in \text{Distr}. d(S) = p\}$, which is a finite set as long as the PLTS $\mathcal{S}$ is finite. Let us define pre $\triangleq \{\text{pre}_a\}_{a \in Act}$ and prob $\triangleq \{\text{prob}_p\}_{p \in \mathcal{W}_\mathcal{S}}$. It is worth noticing that this pair of sets of functions provides an encoding of the PLTS $\mathcal{S}$: pre encodes the transition relation $\to$ of $\mathcal{S}$, while any distribution d in $\mathcal{S}$ can be retrieved through functions in prob. For instance, the support of a distribution $d \in \text{Distr}$ is given by the minimal set of states S such that $d \in \text{prob}_1(S)$, while, for any $s \in \Sigma$, $d(s) = \max \{p \in \mathcal{W}_\mathcal{S} \mid d \in \text{prob}_p(\{s\})\}$.

In the following, the bisimulation problem is stated in terms of forward completeness of pairs of abstract domains $\langle P, \mathcal{P} \rangle \in \text{Part}(\Sigma) \times \text{Part}(\text{Distr})$ w.r.t. the concrete semantic functions prob $\subseteq \wp(\Sigma) \to \wp(\text{Distr})$ and pre $\subseteq \wp(\text{Distr}) \to \wp(\Sigma)$. We first give some useful operational characterizations of pre and prob completeness.

Lemma 4.2 *Let* $\langle P, \mathcal{P} \rangle \in \text{Part}(\Sigma) \times \text{Part}(\text{Distr})$. *Then,* $\langle P, \mathcal{P} \rangle$ *is prob-complete*

(i) *if and only if for any pair of distributions* $d, e \in \text{Distr}$, *if* $e \in \mathcal{P}(d)$ *then* $d \equiv_P e$;
(ii) *if and only if for any block* $B \in P$ *and any distribution* $d \in \text{Distr}$, *the set* $\{e \in \text{Distr} \mid e(B) = d(B)\}$ *is a union of blocks of* $\mathcal{P}$.

Proof Let us prove (i). Assume that $\langle P, \mathcal{P} \rangle$ is prob-complete. Hence, from Definition 4.1 we have that for any $p \in \mathcal{W}_\mathcal{S}$ and for any block $B \in P$, if $d \in \text{prob}_p(B)$ and $\mathcal{P}(d) = \mathcal{P}(e)$ then $e \in \text{prob}_p(B)$. Consider therefore $d, e \in \text{Distr}$ such that $\mathcal{P}(d) = \mathcal{P}(e)$. For any $B \in P$, let us define $p_B \triangleq d(B)$. Since $d \in \text{prob}_{p_B}(B)$, we have that $e \in \text{prob}_{p_B}(B)$, hence $e(B) \geq p_B = d(B)$. Thus, for any $B \in P$, $e(B) \geq d(B)$. Thus, $d(B) = 1 - \sum_{C \in P, C \neq B} d(C) = \sum_{C \in P} e(C) - \sum_{C \in P, C \neq B} d(C) \geq \sum_{C \in P} e(C) - \sum_{C \in P, C \neq B} e(C) = e(B)$, hence $d(B) = e(B)$ for any $B \in P$, namely $d \equiv_P e$.

For the opposite direction, assume that if $\mathcal{P}(d) = \mathcal{P}(e)$ then $d \equiv_P e$ and let us show that $\langle P, \mathcal{P} \rangle$ is prob-complete. Consider $p \in \mathcal{W}_\mathcal{S}$, $X \subseteq \Sigma$, $d \in \text{prob}_p(P(X))$, $e \in \mathcal{P}(d)$ and let us show that $e \in \text{prob}_p(P(X))$. We have that $P(X) = \bigcup_{i \in I} B_i$ for some set of blocks $\{B_i\}_{i \in I} \subseteq P$. Hence, from $d \equiv_P e$, we obtain that $d(\bigcup_i B_i) = e(\bigcup_i B_i)$, so that from $d(\bigcup_i B_i) \geq p$ we get $e(\bigcup_i B_i) \geq p$, namely $e \in \text{prob}_p(P(X))$.

Let now prove (ii). ($\Rightarrow$) Consider $B \in P$ and $d \in \text{Distr}$. It is sufficient to prove that, for any $f, e \in \text{Distr}$, if $f(B) = d(B)$ and $e \in \mathcal{P}(f)$, then $e(B) = d(B)$. Let be $p \triangleq f(B) = d(B)$, then $f \in \text{prob}_p(P(B))$. Hence, by hypothesis, $\mathcal{P}(f) \subseteq \text{prob}_p(P(B))$, then $e \in \text{prob}_p(P(B))$, namely $e(P(B)) = e(B) \geq p$. Let now $Y \triangleq \Sigma \setminus B$ so that $Y = \bigcup_{C \in P, C \neq B} C$. We have that $f(Y) = f(P(Y)) = 1 - p$, so that $f \in \text{prob}_{1-p}(P(Y))$. Hence, by hypothesis, $\mathcal{P}(f) \subseteq \text{prob}_{1-p}(P(Y))$, then $e \in \text{prob}_{1-p}(P(Y))$, namely $e(P(Y)) = e(Y) \geq 1 - p$, from which $e(B) = 1 - e(Y) \leq p$. Hence, $e(B) = p = d(B)$.

($\Leftarrow$) Consider $S \subseteq \Sigma$ and $p \in \mathcal{W}_\mathcal{S}$ and let us show that $\text{prob}_p(P(S))$ is a union of blocks of $\mathcal{P}$. That is, let be $d, e \in \text{Distr}$ such that $d(P(S)) \geq p$ and $\mathcal{P}(d) = \mathcal{P}(e)$, we show that $e(P(S)) \geq p$. Let $P(S) = \bigcup_{i \in I} B_i$ for some set of blocks $\{B_i\}_{i \in I} \subseteq P$. By hypothesis, for any $i \in I$, $\{h \in \text{Distr} \mid h(B_i) = d(B_i)\}$ is a union of blocks of $\mathcal{P}$, hence $e(B_i) = d(B_i)$ for any $i \in I$. Than $e(P(S)) = \sum_{i \in I} e(B_i) = \sum_{i \in I} d(B_i) = d(P(S)) \geq p$. $\square$

Lemma 4.3 *Let* $\langle P, \mathcal{P} \rangle \in \text{Part}(\Sigma) \times \text{Part}(\text{Distr})$. *Then,* $\langle \mathcal{P}, P \rangle$ *is pre-complete*

(i) *if and only if for any* $a \in Act$, $s \xrightarrow{a} d$ *and* $t \in P(s)$ *imply* $t \xrightarrow{a} \mathcal{P}(d)$;
(ii) *if and only if for any block* $C \in \mathcal{P}$ *and for any incoming label* $a \in \text{in}(C)$, $\text{pre}_a(C)$ *is a union of blocks of* P.

Proof Note that since any function pre_a is additive, $\langle \mathcal{P}, P \rangle$ is pre_a-complete if and only if for any block $C \in \mathcal{P}$ and for any incoming label $a \in \text{in}(C)$, we have that $\text{pre}_a(C)$ is a union of blocks of P, which is (ii). Moreover, again by additivity of the function pre_a, we have that $\langle \mathcal{P}, P \rangle$ is pre_a-complete if and only if for any $d \in \text{Distr}$, $\text{pre}_a(\mathcal{P}(d))$ is a union of blocks in P, that is, if and only if for any $d \in \text{Distr}$, $s \in \Sigma$, if $s \xrightarrow{a} \mathcal{P}(d)$ and $P(s) = P(t)$ then $t \xrightarrow{a} \mathcal{P}(d)$, which is equivalent to (i). □

As a direct consequence of the first items of the two lemmas above, we have that a partition $P \in \text{Part}(\Sigma)$ is a bisimulation on $\mathcal{S}$ if and only if the pair of partitions $\langle P, \equiv_P \rangle$ is $\langle \text{prob, pre} \rangle$-complete. In turn, the coarsest bisimulation P_{bis} on $\mathcal{S}$ can be obtained as a forward complete shell of partitions, starting form the top elements $\top_{\text{Part}(\Sigma)} = \{\Sigma\}$ and $\top_{\text{Part(Distr)}} = \{\text{Distr}\}$.

Corollary 4.4 *Let $\mathcal{S} = \langle \Sigma, Act, \rightarrow \rangle$ be a PLTS.*

(i) *$P \in \text{Part}(\Sigma)$ is a bisimulation on $\mathcal{S}$ if and only if $\langle P, \equiv_P \rangle$ is $\langle \text{prob, pre} \rangle$-complete.*

(ii) *Let $\langle P, \mathcal{P} \rangle \in \text{Part}(\Sigma) \times \text{Part(Distr)}$. If $\langle P, \mathcal{P} \rangle$ is $\langle \text{prob, pre} \rangle$-complete then P is a bisimulation on $\mathcal{S}$ and $\mathcal{P} \preceq \equiv_P$.*

Theorem 4.5 $\langle P_{\text{bis}}, \equiv_{P_{\text{bis}}} \rangle = \text{Shell}_{\langle \text{prob,pre} \rangle}(\top_{\text{Part}(\Sigma)}, \top_{\text{Part(Distr)}})$.

Proof Let $\langle P^*, \mathcal{P}^* \rangle = \text{Shell}_{\langle \text{prob,pre} \rangle}(\top_{\text{Part}(\Sigma)}, \top_{\text{Part(Distr)}})$. Since we have that $\langle P^*, \mathcal{P}^* \rangle$ is $\langle \text{prob, pre} \rangle$-complete, by Corollary 4.4(ii), we have that $P^* \preceq P_{\text{bis}}$ and $\mathcal{P}^* \preceq \equiv_{P^*}$. Hence, we also have that $\mathcal{P}^* \preceq \equiv_{P_{\text{bis}}}$. For the opposite direction, by Corollary 4.4(i), we have that $\langle P_{\text{bis}}, \equiv_{P_{\text{bis}}} \rangle$ is $\langle \text{prob, pre} \rangle$-complete, so that, by definition of shell, it turns out that $\langle P_{\text{bis}}, \equiv_{P_{\text{bis}}} \rangle \preceq \langle P^*, \mathcal{P}^* \rangle$, namely $P_{\text{bis}} \preceq P^*$ and $\equiv_{P_{\text{bis}}} \preceq \mathcal{P}^*$. □

4.3 Bisimulation algorithm

By Theorem 4.5, P_{bis} can be computed as a partition shell by instantiating the basic shell algorithm in Fig. 2 to $\mathcal{F} = \{\text{prob}_p\}_{p \in \mathcal{W}_\mathcal{S}}$ and $\mathcal{G} = \{\text{pre}_a\}_{a \in Act}$, and by viewing partitions in $\text{Part}(\Sigma) \times \text{Part(Distr)}$ as abstract domains. This leads to a bisimulation algorithm as described in Fig. 3, called PBis, that takes a PLTS $\mathcal{S}$ as input and initializes and stabilizes a pair of state and distribution partitions $\langle P, \mathcal{P} \rangle$ until it becomes $\langle \text{prob, pre} \rangle$-complete.

Stabilization is obtained by means of two auxiliary functions preStablize() and probStabilize(), that implement respectively Lemma 4.3 and Lemma 4.2. In particular, the function call preStabilize($\langle \mathcal{P}, P \rangle$) refines the state partition P into P' so that $\langle \mathcal{P}, P' \rangle$ is pre-complete. According to Lemma 4.3, in order to get pre-completeness it is sufficient to minimally refine P so that for any block of distributions $C \in \mathcal{P}$, and for any incoming label $a \in \text{in}(C)$, $\text{pre}_a(C)$ is a union of blocks of P. If $\text{pre}_a(C)$ is not a union of blocks of P then $\text{pre}_a(C) \subseteq \Sigma$ is called a splitter of P, and we denote by $\text{Split}(P, \text{pre}_a(C))$ the partition obtained from P by replacing each block $B \in P$ with the sets $B \cap \text{pre}_a(C)$ and $B \setminus \text{pre}_a(C)$, as far as they are nonempty. Notice that when some $\text{pre}_a(C)$ is already a union of blocks of P we have that $\text{Split}(P, \text{pre}_a(C)) = P$, i.e., we also allow no splitting. Hence, preStabilize() refines P by iteratively splitting P w.r.t. $\text{pre}_a(C)$, for all $C \in \mathcal{P}$ and $a \in \text{in}(C)$. On the other hand, the function call probStabilize($\langle P, \mathcal{P} \rangle$) refines the current distribution partition $\mathcal{P}$ into $\mathcal{P}'$ so that $\langle P, \mathcal{P}' \rangle$ is prob-complete. According to Lemma 4.2, $\langle P, \mathcal{P} \rangle$ is prob-complete when for any block $B \in P$ and any distribution $d \in \text{Distr}$, $\{e \in \text{Distr} \mid e(B) = d(B)\}$ is a

```
1 PBis(S) {
2   Initialize();
3   while ¬ (preStable ∧ probStable) do
4     | if ¬ preStable then  probStable := preStabilize(⟨P, P⟩); preStable := true;
5     | if ¬ probStable then  preStable := probStabilize(⟨P, P⟩); probStable := true;
6 }
```

```
1 Initialize() {
2   forall s ∈ Σ do P(s) := Σ;
3   forall d ∈ Distr do P(d) := Distr;
4   preStabilize(⟨P, P⟩); preStable := probStabilize(⟨P, P⟩); probStable := true;
5 }
```

```
1 bool preStabilize(⟨P, P⟩) {
2   P_old := P;
3   forall C ∈ P do
4     | forall a ∈ in(C) do  P := Split(P, pre_a(C));
5   return (P = P_old)
6 }
```

```
1 bool probStabilize(⟨P, P⟩) {
2   P_old := P;
3   forall B ∈ P do
4     | forall d ∈ Distr do  P := Split(P, {e ∈ Distr | e(B) = d(B)});
5   return (P = P_old)
6 }
```

Fig. 3 Bisimulation algorithm PBis

union of blocks of P. Thus, probStabilize() iteratively splits the distribution partition P w.r.t. $\{e \in \text{Distr} \mid e(B) = d(B)\}$, for all $B \in P$ and $d \in \text{Distr}$.

The initialization phase is carried out by the Initialize() function. The two current partitions P and P are initialized with the top elements $\top_{\text{Part}(\Sigma)}$ and $\top_{\text{Part(Distr)}}$, i.e., $\{\Sigma\}$ and $\{\text{Distr}\}$. Moreover, in order to initialize the two Boolean flags probStable and preStable, Initialize() first calls preStabilize() and then calls probStabilize(). Therefore, the initial value of probStable is true, while that of preStable is either true or false depending on whether the prob-stabilization has invalidated the previous pre-stabilization or not.

Theorem 4.6 *For a finite PLTS S, PBis(S) terminates and is correct, i.e., if $\langle P, P \rangle$ is the output of PBis(S) then $P = P_{\text{bis}}$ and $P = \equiv_{P_{\text{bis}}}$.*

Proof A consequence of Theorems 3.1 and 4.5.   □

A strong basis for the implementation of the PBis algorithm is provided by the Baier-Engelen-Majster's two-phased partitioning algorithm [1], which is the standard procedure for computing the bisimulation P_{bis}. Interestingly, this can be essentially viewed as an implementation of the above PBis algorithm, since the two phases of Baier et al.'s algorithm (see [1, Fig. 9]) coincide with our preStabilize() and probStabilize() functions. The only remarkable difference is that instead of using a single partition over all the distributions in Distr, Baier et al.'s algorithm maintains a so-called step partition, namely, a family of partitions $\{M_a\}_{a \in Act}$ such that, for any $a \in Act$, M_a is a partition of the distributions in $\text{post}_a(\Sigma)$, i.e., the distributions that have an incoming edge labeled with a. We do not give here a full account of the implementation of PBis, we rather show how it works on a concrete example.

Example 4.7 Let us illustrate how the algorithm PBis works on the PLTS in Fig. 1, where $\Sigma = \{s_1, s_2, s_3, x_1, \ldots, .x_6, t, u, v, w\}$ and Distr $= \{d_1, d_2, d_3, \delta_t, \delta_u, \delta_v, \delta_w\}$. The call to the Initialize() procedure first sets $P = \{\Sigma\}$ and $\mathcal{P} = \{\text{Distr}\}$. Then the procedure preStabilize() is called and four splitting operations give the following sequence of results:

$$\text{Split}\big(P, \text{pre}_a(\{\text{Distr}\})\big) \Rightarrow P = \big\{\{s_1, s_2, s_3\}, \{x_1, \ldots, .x_6, t, u, v, w\}\big\}$$

$$\text{Split}\big(P, \text{pre}_b(\{\text{Distr}\})\big) \Rightarrow P = \big\{\{s_1, s_2, s_3\}, \{x_1, x_3, x_5\}, \{x_2, x_4, x_6, t, u, v, w\}\big\}$$

$$\text{Split}\big(P, \text{pre}_c(\{\text{Distr}\})\big) \Rightarrow P = \big\{\{s_1, s_2, s_3\}, \{x_1, x_3, x_5\}, \{x_2, x_4\}, \{x_6, t, u, v, w\}\big\}$$

$$\text{Split}\big(P, \text{pre}_d(\{\text{Distr}\})\big) \Rightarrow P = \big\{\{s_1, s_2, s_3\}, \{x_1, x_5\}, \{x_3\}, \{x_2, x_4\}, \{x_6\}, \{t, u, v, w\}\big\}$$

Therefore, at the end of the procedure the state partition P contains six blocks: $B_1 = \{s_1, s_2, s_3\}$, $B_2 = \{x_1, x_5\}$, $B_3 = \{x_3\}$, $B_4 = \{x_2, x_4\}$, $B_5 = \{x_6\}$, $B_6 = \{t, u, v, w\}$. Then the procedure probStabilize() is called and a number of splitting operations are performed. We show in the following only the splitting operations that actually modify the partition $\mathcal{P}$:

$$\text{Split}\big(\mathcal{P}, \{e \mid e(B_2) = d_1(B_2)\}\big) \Rightarrow \mathcal{P} = \big\{\{d_1, d_3\}, \{d_2, \delta_t, \delta_u, \delta_v, \delta_w\}\big\}$$

$$\text{Split}\big(\mathcal{P}, \{e \mid e(B_3) = d_1(B_3)\}\big) \Rightarrow \mathcal{P} = \big\{\{d_1, d_3\}, \{d_2\}, \{\delta_t, \delta_u, \delta_v, \delta_w\}\big\}$$

$$\text{Split}\big(\mathcal{P}, \{e \mid e(B_4) = d_1(B_4)\}\big) \Rightarrow \mathcal{P} = \big\{\{d_1\}, \{d_3\}, \{d_2\}, \{\delta_t, \delta_u, \delta_v, \delta_w\}\big\}$$

Thus, at the end of the procedure the distribution partition $\mathcal{P}$ contains four blocks: $D_1 = \{d_1\}$, $D_2 = \{d_2\}$, $D_3 = \{d_3\}$, $D_4 = \{\delta_t, \delta_u, \delta_v, \delta_w\}$. The main loop of PBis begins with preStable set to false and probStable set to true, and a call to preStabilize() refines P through the following two splitting operations:

$$\text{Split}\big(P, \text{pre}_a(D_1)\big) \Rightarrow P = \big\{\{s_1\}, \{s_2, s_3\}, B_2, B_3, B_4, B_5, B_6\big\}$$

$$\text{Split}\big(P, \text{pre}_a(D_2)\big) \Rightarrow P = \big\{\{s_1\}, \{s_2\}, \{s_3\}, B_2, B_3, B_4, B_5, B_6\big\}$$

A final vacuous call to probStabilize() terminates the computation. Hence, bisimilarity is given by the following partition: $P = \{\{s_1\}, \{s_2\}, \{s_3\}, \{x_1, x_5\}, \{x_3\}, \{x_2, x_4\}, \{x_6\}, \{t, u, v, w\}\}$. In particular, initial states are not bisimilar, while all the final states are bisimilar. Moreover, states x_1 and x_5 as well as x_2 and x_4 are bisimilar since they have a single output transition with the same label and the same target distribution.

5 Simulation as a shell

Let us focus on simulation preorders in PLTSs. We show that the simulation preorder is a complete shell of abstract domains w.r.t. the same operators prob and pre considered above for bisimulation equivalence, whereas the key difference lies in the underlying abstract domains that in this case are preorders, rather than partitions, viewed as abstractions.

5.1 Shells of preorders

Recall that, given any finite set X, $\langle \text{PreOrd}(X), \subseteq, \cup^t, \cap \rangle$ is a lattice, where $R_1 \cup^t R_2$ is the transitive closure of $R_1 \cup R_2$ and the top element is $\top_{\text{PreOrd}(X)} \triangleq X \times X$. Analogously to partitions, any preorder $R \in \text{PreOrd}(X)$ can be viewed as an abstraction of $\langle \wp(X), \subseteq \rangle$, where any set $S \subseteq X$ is approximated by its R-closure $R(S)$. Formally, a preorder $R \in \text{PreOrd}(X)$

can be viewed as the abstract domain closed$(R) \triangleq \{S \subseteq X \mid R(S) = S\}$. Observe that $S \in$ closed(R) iff $S = \bigcup_{i \in I} R(x_i)$ for some set $\{x_i\}_{i \in I} \subseteq X$ and that $\langle$closed$(R), \subseteq, \cup, \cap\rangle$ is a lattice (note that $\varnothing, X \in$ closed(R)). It turns out that closed$(R) \in \langle$Abs$(\wp(X)), \subseteq\rangle$: this means that any set $S \subseteq X$ is approximated by its R-closure, namely by $R(S) \in$ closed(R).

Given a pair of sets of functions $\langle \mathcal{F}, \mathcal{G}\rangle \subseteq (\wp(X) \to \wp(Y)) \times (\wp(Y) \to \wp(X))$, a pair of preorders $\langle R, S\rangle \in$ PreOrd$(X) \times$ PreOrd(Y) is (forward) $\langle \mathcal{F}, \mathcal{G}\rangle$-complete when for any $f \in \mathcal{F}$ and $g \in \mathcal{G}$, if $\langle U, V\rangle \in$ closed$(R) \times$ closed(S) then $\langle f(U), g(V)\rangle \in$ closed$(S) \times$ closed(R).

Definition 5.1 (Forward completeness for preorders) The pair $\langle R, S\rangle$ is f-complete when for any $Z \subseteq X$, $y \in f(R(Z))$ and $y' \in S(y)$, it holds $y' \in f(R(Z))$.

Forward complete shells of preorders are therefore defined as follows: Shell$_{\langle \mathcal{F}, \mathcal{G}\rangle}(\langle R, S\rangle)$ is the largest pair of preorders $\langle R', S'\rangle \subseteq \langle R, S\rangle$ which is $\langle \mathcal{F}, \mathcal{G}\rangle$-complete.

5.2 Simulation on PLTSs

Similarly to the case of bisimulation, simulation can be equivalently expressed in terms of forward completeness w.r.t. prob $= \{$prob$_p\}_{p \in \mathcal{W}_\mathcal{S}}$ and pre $= \{$pre$_a\}_{a \in Act}$.

Lemma 5.2 *Consider a pair $\langle R, \mathcal{R}\rangle \in$ PreOrd$(\Sigma) \times$ PreOrd$($Distr$)$.*

(i) *$\langle R, \mathcal{R}\rangle$ is prob-complete if and only if $e \in \mathcal{R}(d)$ implies $d \leq_R e$;*
(ii) *$\langle \mathcal{R}, R\rangle$ is pre-complete if and only if for any $a \in Act$, $t \in R(s)$ and $s \xrightarrow{a} d$ imply that there exists e such that $t \xrightarrow{a} e$ and $e \in \mathcal{R}(d)$.*

Proof Let us prove (i), that is, $\langle R, \mathcal{R}\rangle$ is prob-complete iff $\mathcal{R} \subseteq \leq_R$. Recall (see [19]) that $d \leq_R e$ iff for any $Z \subseteq \Sigma$, $d(R(Z)) \leq e(R(Z))$.

($\Rightarrow$) Assume that $e \in \mathcal{R}(d)$. For any $Z \subseteq \Sigma$, let us define $p_Z \triangleq d(R(Z))$ so that $d \in$ prob$_{p_Z}(R(Z))$. Thus, by prob-completeness, $e \in$ prob$_{p_Z}(R(Z))$, namely $e(R(Z)) \geq d(R(Z))$. Since this holds for any $Z \subseteq \Sigma$, we have that $d \leq_R e$.

($\Leftarrow$) Consider $Z \subseteq \Sigma$, $p \in \mathcal{W}_\mathcal{S}$, $d \in$ prob$_p(R(Z))$, $e \in \mathcal{R}(d)$ and let us show that $e \in$ prob$_p(R(Z))$. By hypothesis, $d \leq_R e$, so that $e(R(Z)) \geq d(R(Z)) \geq p$, that is, $e \in$ prob$_p(R(Z))$.

Let us turn to (ii). Note that since any function pre$_a$ is additive, $\langle \mathcal{R}, R\rangle$ is pre$_a$-complete if and only if for any $d \in$ Distr, pre$_a(\mathcal{R}(d)) \in$ closed(R), that is, if and only if for any $d \in$ Distr, if $s \xrightarrow{a} \mathcal{R}(d)$ and $t \in R(s)$ then $t \in$ pre$_a(\mathcal{R}(d))$, which is equivalent to (ii). □

Thus, a preorder $R \in$ PreOrd(Σ) is a simulation on $\mathcal{S}$ if and only if the pair $\langle R, \leq_R\rangle$ is $\langle$prob, pre$\rangle$-complete. In turn, the greatest simulation preorder R_{sim} can be obtained as a preorder shell.

Corollary 5.3 *Let $\mathcal{S} = \langle \Sigma, Act, \to\rangle$ be a PLTS.*

(i) *$R \in$ PreOrd(Σ) is a simulation on $\mathcal{S}$ if and only if $\langle R, \leq_R\rangle$ is $\langle$prob, pre$\rangle$-complete.*
(ii) *Let $\langle R, \mathcal{R}\rangle \in$ PreOrd$(\Sigma) \times$ PreOrd$($Distr$)$. If $\langle R, \mathcal{R}\rangle$ is $\langle$prob, pre$\rangle$-complete, then R is a simulation on $\mathcal{S}$ and $\mathcal{R} \subseteq \leq_R$.*

Theorem 5.4 $\langle R_{sim}, \equiv_{R_{sim}}\rangle =$ Shell$_{\langle$prob,pre$\rangle}(\top_{\text{PreOrd}(\Sigma)}, \top_{\text{PreOrd(Distr)}})$.

Proof Analogous to the proof of Theorem 4.5. □

```
1  PSim(𝒮) {
2     Initialize();
3     while ¬ (preStable ∧ probStable) do
4        if ¬ preStable then  probStable := preStabilize(⟨𝓡, R⟩);  preStable := true;
5        if ¬ probStable then  preStable := probStabilize(⟨R, 𝓡⟩);  probStable := true;
6  }
```

Fig. 4 PSim algorithm

```
1  Initialize() {
      // Initialize in
2     forall d ∈ Distr do in(d) := {a ∈ Act | pre_a(d) ≠ ∅};
      // Initialize R and 𝓡
3     forall s ∈ Σ do R(s) := {t ∈ Σ | out(s) ⊆ out(t)};
4     forall d ∈ Distr do 𝓡(d) := {e ∈ Distr | Init_SMF(d, e, R) = true};
      // Initialize Count
5     forall e ∈ Distr do
6        forall a ∈ in(e) do
7           forall x ∈ pre_a(Distr) do
8              Count(x, a, e) := |{d ∈ Distr | x →ᵃ d, d ∈ 𝓡(e), a ∈ in(e)}|;
      // Initialize Remove
9     forall d ∈ Distr do
10       forall a ∈ in(d) do
11          Remove_a(d) := {s ∈ Σ | a ∈ out(s), s ↛ᵃ 𝓡(d)};
      // Initialize Stability Flags
12    probStable := true;
13    if ∃e ∈ Distr, a ∈ in(e) such that Remove_a(e) ≠ ∅ then preStable := false;
14    else preStable := true;
      // Initialize Listener
15    forall x, y ∈ Σ do Listener(x, y) := {(d, e) | x ∈ supp(d), e ∈ supp(e)};
      // Initialize Deleted Arcs
16    Deleted := ∅;
17 }
```

Fig. 5 High-level definition of Initialize() function

6 A new efficient simulation algorithm on PLTSs

We show how a new efficient algorithm for computing simulations in PLTSs, called PSim, can be derived by instantiating the basic shell algorithm to $\mathcal{F} = \{\text{prob}_p\}_{p \in \mathcal{W}_\mathcal{S}}$ and $\mathcal{G} = \{\text{pre}_a\}_{a \in Act}$, and by viewing preorders in PreOrd(Σ) and PreOrd(Distr) as abstract domains. Similarly to the case of bisimulation, PSim, which is described in Fig. 4, takes a PLTS $\mathcal{S}$ as input and initializes and stabilizes a pair of state and distribution preorders $\langle R, \mathcal{R} \rangle \in \text{PreOrd}(\Sigma) \times \text{PreOrd}(\text{Distr})$ until it becomes $\langle \text{prob}, \text{pre} \rangle$-complete.

The stabilization functions, which are given in Fig. 6, refine the preorders according to Lemma 5.2, that is

– the function preStabilize() makes the pair $\langle \mathcal{R}, R \rangle$ pre-complete by refining the state preorder R as long as there exists a transition $s \xrightarrow{a} d$ such that $R(s) \not\subseteq \text{pre}_a(\mathcal{R}(d))$;

1 bool preStabilize($\langle \mathcal{R}, R \rangle$) {
2   Deleted := $\varnothing$;
3   **while** $\exists \, \text{Remove}_a(e) \neq \varnothing$ **do**
4   Remove := $\text{Remove}_a(e)$; $\text{Remove}_a(e) := \varnothing$;
5   **forall** $t \xrightarrow{a} e$ **do**
6   **forall** $w \in \text{Remove}$ **do**
7   **if** $w \in R(t)$ **then** Deleted := Deleted $\cup \{(t, w)\}$; $R(t) := R(t) \smallsetminus \{w\}$;

8   **return** (Deleted $= \varnothing$);
9 }

1 bool probStabilize($\langle R, \mathcal{R} \rangle$) {
2   **forall** $(t, w) \in \text{Deleted}$ **do**
3   **forall** $(d, e) \in \text{Listener}(t, w)$ **such that** $e \in \mathcal{R}(d)$ **do**
4   **if** SMF$(d, e, (t, w)) = \text{false}$ **then**
5 $\mathcal{R}(d) := \mathcal{R}(d) \smallsetminus \{e\}$;
6   **forall** $b \in \text{in}(e) \cap \text{in}(d)$ **do**
7   **forall** $s \xrightarrow{b} e$ **do**
8   Count(s, b, d)--;
9   **if** Count$(s, b, d) = 0$ **then**
10 $\text{Remove}_b(d) := \text{Remove}_b(d) \cup \{s\}$; preStable := false;

11   **return** preStable;
12 }

Fig. 6 Stabilization functions

– the function probStabilize() makes the pair $\langle R, \mathcal{R} \rangle$ prob-complete by iteratively refining the distribution preorder $\mathcal{R}$ as long as there exist $d, e \in \text{Distr}$ such that $e \in \mathcal{R}(d)$ and $d \not\leq_R e$.

The design of these functions allows us to refine the preorders R and $\mathcal{R}$ by following an efficient incremental approach. In particular, preStabilize() refines the preorder R by mimicking the incremental approach of Henzinger et al. [9] simulation algorithm for non-probabilistic LTSs. On the other hand, the function probStabilize() resorts to the incremental approach of Zhang et al. [20] simulation algorithm, and stabilizes the distribution preorder $\mathcal{R}$ by computing sequences of maximum flow problems. More precisely, given a pair of distributions (d, e), successive calls to probStabilize() might repeatedly check whether $d \leq_R e$ where R is the current (new) state preorder. This amounts to repeatedly check whether the maximum flow over the net $\mathcal{N}(d, e, R)$ remains 1 with the current (new) preorder R. Zhang et al. [20] observe that the networks for a given pair (d, e) across successive iterations of their algorithm are very similar, since they differ only by deletion of some edges due to the refinement of R. Therefore, in order to incrementally deal with this sequence of tests, Zhang et al.'s algorithm stores after each iteration the current network $\mathcal{N}(d, e, R)$ together with its maximum flow information, so that at the next iteration, instead of computing the maximum flow of the full new network, one can exploit a so-called preflow algorithm which is initialized with the previous maximum flow function. We do not discuss the details of the preflow algorithm by Zhang et al. [20], since it can be used here as a black box that incrementally solves the sequence of maximum flow problems that arise for similar networks.

PSim relies on a number of global data structures, whose initialization is provided by the Initialize() function, which is described in Fig. 5 at high-level and fully implemented in Fig. 7. The two preorders $R \subseteq \Sigma \times \Sigma$ and $\mathcal{R} \subseteq \text{Distr} \times \text{Distr}$ are stored as Boolean matrices and are initialized in such a way that they are coarser than, respectively, R_{sim} and

```
1   Initialize() {
        //  Initialize In
2       forall d ∈ Distr do
3       │   in(d) := ∅;
4       │   forall a ∈ Act such that pre_a(d) ≠ ∅ do in(d) := in(d) ∪ {a};

        //  Initialize R
5       forall a ∈ Act do  forall x ∈ Σ do mark(a, x) := false;
6       forall d ∈ Distr do
7       │   forall a ∈ in(d) do
8       │   │   forall x ∈ pre_a(d) do mark(a, x) := true;

9       forall d ∈ Distr do
10      │   forall a ∈ in(d) do
11      │   │   forall x ∈ pre_a(d) do
12      │   │   │   forall y ∈ Σ such that mark(a, y) = false do R(x, y) := false;

        //  Initialize R
13      forall d, e ∈ Distr do  R(d, e) := Init_SMF(d, e, R)
        //  Initialize Count
14      forall d ∈ Distr do
15      │   forall a ∈ in(d) do
16      │   │   forall x ∈ pre_a(d) do
17      │   │   │   forall e ∈ Distr such that pre_a(e) ≠ ∅ do Count(x, a, e) := 0;

18      forall d ∈ Distr do
19      │   forall a ∈ in(d) do
20      │   │   forall x ∈ pre_a(d) do
21      │   │   │   forall e ∈ Distr such that R(e, d) ∧ pre_a(e) ≠ ∅ do Count(x, a, e)++;

        //  Initialize Remove
22      forall d ∈ Distr do
23      │   forall a ∈ in(d) do
24      │   │   Remove_a(d) := ∅;
25      │   │   forall x ∈ Σ do
26      │   │   │   if Count(x, a, d) = 0 then Remove_a(d) := Remove_a(d) ∪ {x};

        //  Initialize Stability Flags
27      probStable := true;
28      if ∃e ∈ Distr, a ∈ in(e) such that Remove_a(e) ≠ ∅ then preStable := false;
29      else preStable := true;
        //  Initialize Listener
30      forall x, y ∈ Σ do Listener(x, y) := ∅;
31      forall d, e ∈ Distr do
32      │   forall x ∈ supp(d), y ∈ supp(e) do Listener(x, y) := Listener(x, y) ∪ {(d, e)};
        //  Initialize Deleted Arcs
33      Deleted := ∅;
34  }
```

Fig. 7 Implementation of Initialize() function

$\leq_{R_{sim}}$. In particular, the initial preorder R is coarser than R_{sim} since if $s \xrightarrow{a} d$ and $t \xnrightarrow{a}$ then $t \notin R_{sim}(s)$. Moreover, line 4 of Fig. 5 initializes $\mathcal{R}$ so that $\mathcal{R} = \leq_R$: this is done by calling the function Init_SMF(d, e, R) which in turn calls the preflow algorithm to check whether $d \leq_R e$, and in case this is true, stores the network $\mathcal{N}(d, e, R)$ in order to reuse it in later calls to probStabilize(). The additional data structures used by PSim come from the incremental refinement methods used in [9] and [20]. Actually, as in [9], for any distribution e and for

any incoming action $a \in \text{in}(e)$, we store and maintain a set

$$\text{Remove}_a(e) \triangleq \left\{ s \in \Sigma \mid s \xrightarrow{a}, \ s \xnrightarrow{a} \mathcal{R}(e) \right\}$$

that is used to prune the relation R to get pre-completeness (lines 5–7 of preStabilize()). The $\{\text{Count}(s, a, e)\}_{e \in \text{Distr}, a \in \text{in}(e), s \in \text{pre}_a(\text{Distr})}$ table records in any entry $\text{Count}(s, a, e)$ the number of a-transitions from state s to a distribution in $\mathcal{R}(e)$, so that it can be used to efficiently refill the remove sets (line 10 of probStabilize()), since it allows to test whether $s \xnrightarrow{a} \mathcal{R}(e)$ in $O(1)$ by checking whether $\text{Count}(s, a, e)$ is equal to 0. Moreover, in order to get an efficient refinement also for the distribution preorder $\mathcal{R}$, likewise to Zhang et al.'s algorithm, for any pair of states (x, y) we compute and store a set

$$\text{Listener}(x, y) \triangleq \left\{ (d, e) \in \text{Distr} \times \text{Distr} \mid x \in \text{supp}(d) \wedge y \in \text{supp}(e) \right\}$$

that contains all the pairs of distributions (d, e) such that the network $\mathcal{N}(d, e, R)$ could contain the edge (x, y), i.e., $\text{Listener}(x, y)$ records the networks that are affected when the pair of states (x, y) is removed from the preorder R. Indeed, these sets are used in probStabilize() to recognize the pairs (d, e) that have been affected by the refinement of R due to the previous call of preStabilize() (lines 2–3 of probStabilize()).

At the end of initialization, the probStable flag is set to true (due to the initialization of $\mathcal{R}$ as $\leq_R$), whereas the preStable flag is set to false if a nonempty remove set exists. The main loop of PSim then repeatedly calls the stabilization functions until the pair $\langle R, \mathcal{R} \rangle$ becomes $\langle \text{prob}, \text{pre} \rangle$-complete. More precisely, a call to preStabilize(): (i) refines the relation R in such a way that if $t \xrightarrow{a} e$ then $R(t)$ is pruned to $R(t) \smallsetminus \text{Remove}_a(e)$, (ii) empties all the Remove sets and collects all the pairs removed from R into the set Deleted, and (iii) sets the probStable flag to false when R has changed. On the other hand, a call to probStabilize() exploits the sets Deleted and Listener to determine all the networks $\mathcal{N}(d, e, R)$ that may have been affected by the refinement of R due to preStabilize(). For any pair (t, w) that has been removed from R, the call $\text{SMF}(d, e, (t, w))$ at line 4 removes the edge (t, w) from the network for (d, e) and calls the preflow algorithm to check whether it still has a maximum flow equals to 1. Then, if this is not the case, e is removed from $\mathcal{R}(d)$. Notice that such a pruning may induce an update of some $\text{Remove}_b(d)$ set, which in turn triggers a further call of preStabilize() by setting the preStable flag to false.

Example 6.1 Let us illustrate how the algorithm PSim works on the PLTS in Fig. 1, where $\Sigma = \{s_1, s_2, s_3, x_1, \ldots, .x_6, t, u, v, w\}$ and $\text{Distr} = \{d_1, d_2, d_3, \delta_t, \delta_u, \delta_v, \delta_w\}$. The call to Initialize() yields the following preorders and remove sets:

$$
\begin{aligned}
&R(x_1) = R(x_5) = \{x_1, x_3, x_5\} &\quad &R(s_1) = R(s_2) = R(s_3) = \{s_1, s_2, s_3\} \\
&R(x_2) = R(x_4) = \{x_2, x_4\} &\quad &R(t) = R(u) = R(v) = R(w) = \Sigma \\
&R(x_3) = \{x_3\} &\quad &R(x_6) = \{x_6, x_3\} \\[4pt]
&\mathcal{R}(d_1) = \{d_1, d_2\} &\quad &\mathcal{R}(d_2) = \{d_2\} \\
&\mathcal{R}(d_3) = \{d_3\} &\quad &\mathcal{R}(\delta_t) = \mathcal{R}(\delta_u) = \mathcal{R}(\delta_v) = \mathcal{R}(\delta_w) = \text{Distr} \\[4pt]
&\text{Remove}_a(d_1) = \{s_3\} &\quad &\text{Remove}_a(d_2) = \{s_1, s_3\} \\
&\text{Remove}_a(d_3) = \{s_1, s_2\} &\quad &\text{Remove}_b(\delta_t) = \text{Remove}_c(\delta_u) = \varnothing \\
&\text{Remove}_d(\delta_v) = \text{Remove}_d(\delta_w) = \varnothing
\end{aligned}
$$

The main loop of PSim begins with preStable set to false and probStable set to true, and a call to preStabilize() refines R of s_1, s_2 and s_3 to: $R(s_1) = \{s_1, s_2\}$, $R(s_2) = \{s_2\}$, $R(s_3) = \{s_3\}$.

A final vacuous call to probStabilize() terminates the computation. Hence, R_{sim} is as follows:

$$R_{\text{sim}}(s_1) = \{s_1, s_2\} \quad R_{\text{sim}}(s_2) = \{s_2\}$$
$$R_{\text{sim}}(s_3) = \{s_3\} \quad R_{\text{sim}}(x_1) = R_{\text{sim}}(x_5) = \{x_1, x_3, x_5\}$$
$$R_{\text{sim}}(x_3) = \{x_3\} \quad R_{\text{sim}}(x_2) = R_{\text{sim}}(x_4) = \{x_2, x_4\}$$
$$R_{\text{sim}}(x_6) = \{x_6, x_3\} \quad R_{\text{sim}}(t) = R_{\text{sim}}(u) = R_{\text{sim}}(v) = R_{\text{sim}}(w) = \Sigma$$

In particular, we have that s_2 simulates s_1 while s_1 does not simulate s_2, and s_2 does not simulate s_3.

6.1 Correctness

The correctness of PSim comes as a consequence of Theorems 3.1 and 5.4 and the fact that the procedures in Fig. 6 correctly stabilize the preorders R and $\mathcal{R}$.

Lemma 6.2 (Correctness of preStabilize()) *Let* $\langle R, \mathcal{R} \rangle \in \text{PreOrd}(\Sigma) \times \text{PreOrd}(\text{Distr})$ *and* $\langle R', \mathcal{R}' \rangle$ *be the pair of preorders at the exit of a call to* preStabilize($\langle \mathcal{R}, R \rangle$). *Then,* $\mathcal{R}' = \mathcal{R}$ *and* $R' \subseteq R$ *is such that for any* $s, t \in \Sigma$, $d \in \text{Distr}$, $a \in \text{Act}$, *if* $s \xrightarrow{a} d$ *and* $t \in R'(s)$ *then* $t \xrightarrow{a} \mathcal{R}(d)$, *i.e.,* $\langle \mathcal{R}', R' \rangle$ *is* pre-*complete.*

Proof preStabilize($\langle \mathcal{R}, R \rangle$) does not modify the distribution preorder $\mathcal{R}$, hence $\mathcal{R}' = \mathcal{R}$. Consider $s, t \in \Sigma$, $d \in \text{Distr}$ and $a \in \text{Act}$ with $s \xrightarrow{a} d$ and $t \in R'(s)$. Since R' is a refinement of the initial state preorder R, we have that $a \in \text{out}(t)$, so that there exists a distribution e such that $t \xrightarrow{a} e$. Moreover, $e \in \mathcal{R}(d)$, otherwise t would belong to $\text{Remove}_a(d)$, which instead is empty, because at the exit of preStabilize($\langle \mathcal{R}, R \rangle$) every remove set is empty. By Lemma 5.2, $\langle \mathcal{R}', R' \rangle$ is therefore pre-complete. □

Lemma 6.3 (Correctness of probStabilize()) *Let* $\langle R, \mathcal{R} \rangle \in \text{PreOrd}(\Sigma) \times \text{PreOrd}(\text{Distr})$ *and* $\langle R', \mathcal{R}' \rangle$ *be the pair of preorders at the exit of a call to* probStabilize($\langle R, \mathcal{R} \rangle$), *where* $\mathcal{R} = \leq_{R \cup \text{Deleted}}$. *Then* $R' = R$ *and* $\mathcal{R}' \subseteq \mathcal{R}$ *is such that for any* $d, e \in \text{Distr}$, *if* $e \in \mathcal{R}'(d)$, *then* $d \leq_R e$, *i.e.,* $\langle R', \mathcal{R}' \rangle$ *is* prob-*complete.*

Proof probStabilize($\langle R, \mathcal{R} \rangle$) does not modify the state preorder R, hence $R' = R$. Consider $e \in \mathcal{R}'(d)$, so that $e \in \mathcal{R}(d)$. By hypothesis, we have that the maximal flow in the network $\mathcal{N}(d, e, R \cup \text{Deleted})$ is 1. If $\text{Deleted} \cap (\text{supp}(d) \times \text{supp}(e)) = \varnothing$, then $\mathcal{N}(d, e, R \cup \text{Deleted}) = \mathcal{N}(d, e, R)$, and therefore $d \leq_R e$. Otherwise, the fact that $e \in \mathcal{R}'(d)$ means that probStabilize() has (repeatedly) checked (at line 4) that $\mathcal{N}(d, e, R)$ remained 1, so that $d \leq_R e$. Thus, by Lemma 5.2, $\langle R', \mathcal{R}' \rangle$ is prob-complete. □

Theorem 6.4 (Correctness of PSim) *Let* $\mathcal{S}$ *be a finite PLTS. Then,* PSim($\mathcal{S}$) *always terminates with output* $\langle R_{\text{sim}}, \leq_{R_{\text{sim}}} \rangle$.

Proof The fact that PSim terminates on a finite input PLTS depends on the fact that at each iteration probStabilize() and/or preStabilize() refine $\mathcal{R}$ and/or R. The correctness of the output comes from Theorems 3.1 and 5.4 together with the above Lemmata 6.2 and 6.3. □

6.2 Complexity

Given a PLTS $\mathcal{S} = \langle \Sigma, \text{Act}, \rightarrow \rangle$, the complexity bounds of PSim($\mathcal{S}$) are given in terms of the following sizes:

- $|\text{Distr}| = |\bigcup_{a \in Act} \text{post}_a(\Sigma)|$ is the number of distinct distributions appearing as target of some transition in $\mathcal{S}$.
- Let us define

$$p \triangleq \sum_{d \in \text{Distr}} |\text{supp}(d)| \quad \text{and} \quad m \triangleq \sum_{s \in \Sigma} \sum_{a \in \text{out}(s)} \sum_{d \in \text{post}_a(s)} (|\text{supp}(d)| + 1)$$

where the size m has been defined in [20, Sect. 2]. Thus, p represents the full size of $\text{Distr} = \text{post}(\Sigma)$, being the number of states that appear in the support of some distribution in Distr. On the other hand, m represents the number of transitions in the PLTS $\mathcal{S}$, that is, the number of transitions "from states to states", where a "state transition" (s, t) is taken into account when $s \xrightarrow{a} d$ and $t \in \text{supp}(d)$. Notice that $p \leq |\Sigma||\text{Distr}|$ and $m \leq |\Sigma||Act||\text{Distr}|$. The key point to remark here is that $p \leq m$, since the number of "states" of $\mathcal{S}$ are always less than or equal to the number of "state transitions" in $\mathcal{S}$. As an example, in the PLTS of Fig. 1, we have that $p = 10 = \Sigma_{i=1,2,3}|\text{supp}(d_i)| + \text{supp}(\delta_t) + \text{supp}(\delta_u) + \text{supp}(\delta_v) + \text{supp}(\delta_w)$, while $m = 23 = \text{supp}(d_1) + 1 + \text{supp}(d_2) + 1 + \text{supp}(d_3) + 1 + \text{supp}(\delta_t) + 1 + \text{supp}(\delta_u) + 1 + \text{supp}(\delta_t) + 1 + \text{supp}(\delta_v) + 1 + \text{supp}(\delta_u) + 1 + \text{supp}(\delta_t) + 1 + \text{supp}(\delta_w) + 1$.

Lemma 6.5 (Complexity of SMF) *All the calls to* $\text{SMF}(d, e, \ldots)$ *relative to a pair of distributions* d, e, *including the first call* $\text{Init\_SMF}(d, e, R)$, *overall take* $O(|\text{supp}(d)||\text{supp}(e)|^2)$ *time.*

Proof See [20, Lemma 4.4]. □

Theorem 6.6 (Complexity of PSim) *Let* $\mathcal{S}$ *be a finite PLTS.* $\text{PSim}(\mathcal{S})$ *runs in* $O(|\Sigma|(p^2 + |\Sigma|) + |\rightarrow|(|\Sigma| + |\text{Distr}|))$-*time and* $O(p^2 + |\Sigma|^2 + |\rightarrow|(|\Sigma| + |\text{Distr}|))$-*space.*

Proof Let us first discuss how we represent the input PLTS $\mathcal{S}$. Let $s_1, \ldots, s_n$ be a fixed enumeration of the states in Σ, let $a_1, \ldots, a_k$ be a fixed enumeration of the labels in Act and let $d_1, \ldots, d_m$ be a fixed enumeration of the distributions in $\text{Distr} = \bigcup_{a \in Act} \text{post}_a(\Sigma)$. We assume that any distribution $d \in \text{Distr}$ is represented as a record with two components:

- an array $[\text{pre}_{a_1}, \ldots, \text{pre}_{a_k}]$, where the (possibly empty) i-th entry pre_{a_i} is a pointer to the list of states s such that $s \xrightarrow{a_i} d$.
- a list of pairs $(s_1, d(s_1)), \ldots, (s_r, d(s_r))$ that enumerates the states in the support of d together with their (non-zero) probability.

Time Complexity. The time complexities of the functions called by PSim are as follows, where the cost of Initialize() refers to the implementation given in Fig. 7.

Initialize() takes $O(p^2 + |\Sigma|^2 + |\rightarrow|(|\Sigma| + |\text{Distr}|))$ time, as detailed below:

- The initialization of the in(·) sets takes $O(|\text{Distr}||Act|)$ time since, given a distribution d, the test $\text{pre}_a(d) \neq \varnothing$ is done in $O(1)$ since it is sufficient to test if the a-component of the array stored in the distribution d is non-null.
- Initialization of R and $\mathcal{R}$: lines 5–12 takes $O(|\Sigma||Act| + |\rightarrow| + |\Sigma||\rightarrow|)$ time. while the cost of the calls to Init_SMF() will be considered together with the global cost of all the calls to the function SMF() in probStabilize().
- Initialization of the Count table: the cost of lines 14–21 is $O(|\text{Distr}||\rightarrow|)$ since, as discussed above, the test $\text{pre}_a(e) \neq \varnothing$ takes constant time.
- Initializing the Remove sets (lines 22–26) takes $O(|\Sigma||\rightarrow|)$ time.

– Initializing the stability flags (lines 27–29) takes $O(|\rightarrow|)$ time.
– Initialization of the Listener sets: line 30 takes $O(|\Sigma|^2)$, while the cost of lines 31–32 is $\sum_{d\in\text{Distr}}\sum_{e\in\text{Distr}}|\operatorname{supp}(d)||\operatorname{supp}(e)|\leq p^2$.

All the calls to preStabilize() globally cost $O(|\Sigma||\rightarrow|+|\Sigma|^2)$ time, similarly to what happens in Henzinger et al.'s simulation algorithm [9]:

– line 4: given a pair (e,a), the overall cost of this line is in $O(|\text{pre}_a(\text{Distr})|)$ since $\text{Remove}_a(e)\subseteq\text{pre}_a(\text{Distr})$ and when the same pair appears as pivot of different iterations, say i and j, the sets $\text{Remove}_a^i(e)$ and $\text{Remove}_a^j(e)$ relative to the different iterations i and i are disjoint. Hence, it turns out that the overall cost of line 4 is $\sum_{e\in\text{Distr}}\sum_{a\in\text{in}(e)}|\text{pre}_a(\text{Distr})|$, which is in $O(|\Sigma||\rightarrow|)$.
– Since the sets $\text{Remove}_a^i(e)$ and $\text{Remove}_a^j(e)$ relative to two different iterations i and j are disjoint, we have that any transition $t\xrightarrow{a}e$ in line 5 is traversed at most $|\bigcup_i\{w\mid w\in\text{Remove}_a^i(e)\}|$ times, namely, at most $|\{w\mid a\in\text{out}(w)\}|$ times. Hence, the overall cost of lines 5–6-and the test at line 7 is $\sum_{w\in\Sigma}\sum_{a\in\text{out}(w)}|\xrightarrow{a}|$ and therefore is in $O(|\Sigma||\rightarrow|)$.
– The body of the if statement at line 7 globally takes $|\Sigma|^2$ time since the pairs (t,w) such that the if-test is positive are globally pairwise disjoint.

The cost of all the calls to probStabilize() is in $O(p^2|\Sigma|+|\text{Distr}||\rightarrow|)$ time, as detailed below:

– Let us first consider the cost of all the calls to SMF() and Init_SMF(). By Lemma 6.5, for all the pairs (d,e), with $d,e\in\text{Distr}$, the cost of all the calls to SMF() and Init_SMF() is $\sum_{d\in\text{Distr}}\sum_{e\in\text{Distr}}|\operatorname{supp}(d)||\operatorname{supp}(e)|^2$, which is in $O(p^2|\Sigma|)$ since for any distribution e it holds that $|\operatorname{supp}(e)|\leq|\Sigma|$.
– The same pair (d,e) may appear in at most $|\operatorname{supp}(d)||\operatorname{supp}(e)|$ different Listener sets. The set of pairs $(t,w)\in\text{Deleted}$ are pairwise disjoint throughout all the calls to probStabilize(), thus the overall cost of lines 2–3 is $\sum_{d\in\text{Distr}}\sum_{e\in\text{Distr}}|\operatorname{supp}(d)||\operatorname{supp}(e)|=p^2$.
– To estimate the overall cost of lines 5–10 of probStabilize(), observe that the test at line 4 is positive at most once for every pair d,e, because after a positive test e is removed from $\mathcal{R}(d)$ and never put back. Hence, the overall cost of lines 5–10 is $\sum_{d\in\text{Distr}}\sum_{e\in\text{Distr}}(1+\sum_{b\in\text{in}(e)\cap\text{in}(d)}|\text{pre}_b(e)|)$, which is in $O(|\text{Distr}||\rightarrow|)$.

Finally, notice that the test of the main while loop of PSim is performed $|\Sigma|^2$ times since the relation R can be refined at most $|\Sigma|^2$ times. Thus, summing up, it turns out that the time complexity of PSim is in $O(|\Sigma|(p^2+|\Sigma|)+|\rightarrow|(|\Sigma|+|\text{Distr}|))$.

Space Complexity. PSim relies on the following data structures:

– the Boolean table $\{\text{mark}(a,x)\}_{a\in Act,x\in\Sigma}$ that takes $O(|Act||\Sigma|)$ space;
– the in lists of labels, that take $O(|\text{Distr}||Act|)$ space;
– the networks $\mathcal{N}(d,e,R)$ that are updated at each iteration. According to [20], the space needed to store these networks is $\sum_{d\in\text{Distr}}\sum_{e\in\text{Distr}}|\operatorname{supp}(d)||\operatorname{supp}(e)|\leq p^2$;
– the two Boolean matrixes $\{R(s,t)\}_{s,t\in\Sigma}$ and $\{\mathcal{R}(d,e)\}_{d,e\in\text{Distr}}$ that take, respectively, $O(|\Sigma|^2)$ and $O(|\text{Distr}|^2)$ space.
– the integer table $\{\text{Count}(s,a,e)\}_{e\in\text{Distr},\,a\in\text{in}(e),\,s\in\text{pre}_a(\text{Distr})}$ and the lists of states $\{\text{Remove}_a(e)\}_{e\in\text{Distr},a\in\text{in}(e)}$ take, respectively, $O(|\text{Distr}||\rightarrow|)$ and $O(|\Sigma||\rightarrow|)$ space;
– the sets $\{\text{Listener}(x,y)\}_{x,y\in\Sigma}$ take p^2 space because, as above, the same pair (d,e) may appear in at most $|\operatorname{supp}(d)||\operatorname{supp}(e)|$ different Listener sets;
– the set of deleted arcs in Deleted takes $O(|\Sigma|^2)$ space.

Thus, the overall space complexity of PSim is in $O(p^2+|\Sigma|^2+|\rightarrow|(|\Sigma|+|\text{Distr}|))$. $\quad\square$

Notice that $|\Sigma|, |\rightarrow|, |\text{Distr}| \leq m$ and recall that $p \leq m$. Hence, $|\Sigma|(p^2 + |\Sigma|) \leq |\Sigma|m^2$ and $|\rightarrow|(|\Sigma| + |\text{Distr}|) \leq 2m^2$. As a consequence, PSim turns out to be at least as efficient than the most efficient probabilistic simulation algorithm in literature, that is Zhang et al.'s algorithm [20], that runs in $O(|\Sigma|m^2)$-time. This also holds for space complexity, since Zhang et al.'s algorithm [20] has a $O(m^2)$ space bound, while for PSim we have that $p^2 + |\Sigma|^2 + |\rightarrow|(|\Sigma| + |\text{Distr}|) \leq 4m^2$. Our reduction from the size m to p, that is from the size of the "state transitions" to the size of the "state" space, basically depends on the fact that in Zhang et al.'s algorithm the same test $d \not\leq_R e$ is repeated for every pair of states (s_i, t_i) such that $s_i \in \text{pre}_a(d), t_i \in \text{pre}_a(e)$, whereas in PSim once the test $d \not\leq_R e$ has been performed, every state t_i is removed from $R(s_i)$. Let us observe that when the input PLTS $\mathcal{S}$ degenerates to a LTS, a call to the function SMF() can be executed in constant time, so that PSim runs in $O(|\Sigma||\rightarrow| + |\Sigma|^2)$-time, essentially reducing to Henzinger et al.'s nonprobabilistic simulation algorithm [9]. It is also worth observing that Zhang et al.'s algorithm relies on a positive strategy that at each iteration i computes the pairs (s_i, t_i) such that $t_i \in R_i(s_i)$, whereas PSim follows a dual, negative, strategy that removes from R_i the pairs (s_i, t_i) such that $t_i \notin R_i(s_i)$.

7 Future work

We have shown how abstract interpretation can be fruitfully applied in the context of behavioral relations between probabilistic processes. We focused here on bisimulation/simulation relations on PLTSs and we proved how efficient algorithms that compute these behavioral relations can be systematically derived. As future work, we plan to investigate how this abstract interpretation approach can be adapted to characterize the weak variants of bisimulation/simulation and the so-called probabilistic bisimulations/simulations on PLTSs [16]. We also plan to apply a coarsest partition refinement approach to design a "symbolic" version of our PSim simulation algorithm. Analogously to the symbolic algorithm by Ranzato and Tapparo [13, 15] for nonprobabilistic simulation, the basic idea is to symbolically represent the relations R on states and $\mathcal{R}$ on distributions through partitions (of states and distributions) and corresponding relations between blocks of these relations. It is worth noting that this partition refinement approach has been already applied by Zhang [18] to design a space-efficient simulation algorithm for PLTSs.

References

1. Baier C, Engelen B, Majster-Cederbaum M (2000) Deciding bisimilarity and similarity for probabilistic processes. J Comput Syst Sci 60:187–231
2. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc 4th ACM POPL, pp 238–252
3. Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: Proc 6th ACM POPL, pp 269–282
4. Crafa S, Ranzato F (2011) Probabilistic bisimulation and simulation algorithms by abstract interpretation. In: Proc ICALP'11. Springer LNCS, vol 6756, pp 295–306
5. Desharnais J (1999) Labelled Markov processes. PhD thesis, McGill Univ
6. Desharnais J, Edalat A, Panangaden P (2002) Bisimulation for labelled Markov processes. Inf Comput 179:163–193

7. Giacobazzi R, Quintarelli E (2001) Incompleteness, counterexamples and refinements in abstract model checking. In: Proc 8th SAS. Springer LNCS, vol 2126, pp 356–373
8. Giacobazzi R, Ranzato F (1997) Refining and compressing abstract domains. In: Proc 24th ICALP. Springer LNCS, vol 1256, pp 771–781
9. Henzinger MR, Henzinger TA, Kopke PW (1995) Computing simulations on finite and infinite graphs. In: Proc 36th FOCS, pp 453–462
10. Larsen KG, Skou A (1991) Bisimulation through probabilistic testing. Inf Comput 94(1):1–28
11. Paige R, Tarjan RE (1987) Three partition refinement algorithms. SIAM J Comput 16(6):973–989
12. Parma A, Segala R (2007) Logical characterizations of bisimulations for discrete probabilistic systems. In: Proc FOSSACS'07. Springer LNCS, vol 4423, pp 287–301
13. Ranzato F, Tapparo F (2007) A new efficient simulation equivalence algorithm. In: Proc IEEE LICS'07, pp 171–180
14. Ranzato F, Tapparo F (2007) Generalized strong preservation by abstract interpretation. J Log Comput 17(1):157–197
15. Ranzato F, Tapparo F (2010) An efficient simulation algorithm based on abstract interpretation. Inf Comput 208(1):1–22
16. Segala R, Lynch N (1995) Probabilistic simulations for probabilistic processes. Nord J Comput 2(2):250–273
17. van Glabbeek RJ, Smolka S, Steffen B, Tofts C (1990) Reactive, generative and stratified models for probabilistic processes. In: Proc IEEE LICS'90, pp 130–141
18. Zhang L (2008) A space-efficient probabilistic simulation algorithm. In: Proc CONCUR'08. Springer LNCS, vol 5201, pp 248–263
19. Zhang L (2009) Decision algorithms for probabilistic simulations. PhD thesis, Univ des Saarlandes
20. Zhang L, Hermanns H, Eisenbrand F, Jansen DN (2008) Flow faster: efficient decision algorithms for probabilistic simulations. Log Methods Comput Sci 4(4)