Deciding Bisimilarity and Similarity for Probabilistic Processes*

Christel Baier
Universität Bonn, Germany
baier@cs.uni-bonn.de

Bettina Engelen†, Mila Majster-Cederbaum
Universität Mannheim, Germany
{bengelen,mcb}@pi2.informatik.uni-mannheim.de

August 20, 1999

*A preliminary version of this paper has appeared in [4].

†Supported by Deutsche Forschungsgesellschaft (DFG MA 794/3-1)

Proposed running head: Deciding bisimilarity and similarity

Contact address:

Christel Baier
Universität Bonn
Institut für Informatik I
Römerstr. 164
D-53117 Bonn, Germany

EMail: baier@cs.uni-bonn.de
Tel: ++49-228-734333
Fax: ++49-228-734321

Abstract

This paper deals with probabilistic and nondeterministic processes represented by a variant of labelled transition systems where any outgoing transition of a state s is augmented with probabilities for the possible successor states. Our main contribution are algorithms for computing the bisimulation equivalence classes as introduced by Larsen & Skou [44] and the simulation preorder à la Segala & Lynch [57]. The algorithm for deciding bisimilarity is based on a variant of the traditional partitioning technique [43, 51] and runs in time $\mathcal{O}(mn(\log m + \log n))$ where m is the number of transitions and n the number of states. The main idea for computing the simulation preorder is the reduction to maximum flow problems in suitable networks. Using the method of Cheriyan, Hagerup & Mehlhorn [15] for computing the maximum flow, the algorithm runs in time $\mathcal{O}((mn^6 + m^2n^3)/\log n)$. Moreover, we show that the network-based technique is also applicable to compute the simulation-like relation of [40] in fully probabilistic systems (a variant of ordinary labelled transition systems where the non-determinism is totally resolved by probabilistic choices).

1 Introduction

Labelled transition systems (LTSs) together with a variety of widely accepted equivalence relations (e.g. bisimulation [47, 52], trace or failure equivalence [13]) and preorders (e.g. simulation [48] or testing preorders [50]) have proved to be very useful for modelling and analyzing concurrent processes. Typically, the equivalences or preorders are defined as relations on the state space of a LTS but they can be extended for the comparison of two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ (by taking the disjoint union of the state spaces and comparing the initial states of $\mathcal{P}_1$ and $\mathcal{P}_2$ in the composed structure); thus, yielding a formalization of when a process correctly implements another one. Intuitively, the equivalences can be viewed as a notion *process equality* stating when two processes have the same "behaviours". (Hence, equivalent processes satisfy exactly the same properties of any formalism for which the satisfaction relation refers to a corresponding notion of "behaviour".) The preorders are "uni-directed" and lead to a notion of "behave as good as". Many preorders $\preceq$ allow for an interpretation in the style $\mathcal{P}_1 \preceq \mathcal{P}_2$ iff any possible behaviour of $\mathcal{P}_1$ is also a possible behaviour of $\mathcal{P}_2$. Thus, any safety property (which states the absence of certain "bad" behaviours) that holds for $\mathcal{P}_2$ also holds for $\mathcal{P}_1$. The kernel $\approx = \preceq \cap \preceq^{-1}$ of a preorder $\preceq$ is then an equivalence that identifies only those processes which satisfy the same safety properties.

Satisfying certain congruence properties, e.g. for a certain type of parallel composition operator $\|$, the equivalences and preorders support the use of LTSs for the design by *stepwise refinement*. For instance, when dealing with an equivalence, one might replace a high-level specification $\mathcal{Q}$ of a certain module of a composed program $\mathcal{P} = \ldots \| \mathcal{Q} \| \ldots$ by an equivalent lower-level specification $\mathcal{Q}'$; thus obtaining a refined program $\mathcal{P}' = \ldots \| \mathcal{Q}' \| \ldots$ which is equivalent to the original (more abstract) program $\mathcal{P}$ and hence satisfies the same properties as $\mathcal{P}$. Proceeding in this way, a low-level formulation (which might be closed to a real implementation) of a complex program can be derived from a high-level description (the specification) by successively replacing module specifications by refined (equivalent) ones.

Moreover, the equivalences play a crucial role for the system analysis by *abstraction*. This means the replacement of the original state space S by the (possibly much smaller) quotient space $S/\approx$ with respect to an appropriate equivalence $\approx$.

In particular, algorithmic methods for checking whether an equivalence or preorder can be established between two processes or for computing the quotient space with respect to an equivalence (see e.g. [43, 51, 12, 20, 31, 25, 38]) are of great importance as they can serve as basis for an automatic verification tool. For practical applications, the time and space efficiency of such methods is a crucial aspect. As most interesting properties can be specified in the linear time framework, the linear time relations, such as trace equivalence or trace containment, are the most important ones as they yield the "best abstraction" under all relations that preserve linear time properties. However, the verification problem for them is quite expensive (PSPACE-complete [43]). A widespread technique to overcome the limitations of the linear time framework due to these complexity results is the switch to finer relations of the branching time spectrum (e.g. bisimulation equivalence which refines trace equivalence or the simulation preorder which is finer than trace containment) for

which efficient decision procedures exist. For instance, for a LTS with n states and m transitions, bisimilarity can be decided in time $\mathcal{O}(m \log n)$ [51], the simulation preorder can be computed in time $\mathcal{O}(mn)$ [38].

In recent years, many researchers have focussed on reasoning about probabilistic phenomena that might occur e.g. in processes that are designed on the basis of a randomized algorithm or processes with uncertain components where probabilities can be used to specify failure rates. In the literature, a variety of probabilistic models has been proposed. Most of them are probabilistic variants of transition systems. These models can be classified with respect to their treatment of non-determinsm.

Several authors deal with *fully probabilistic* models (e.g. the *generative* model in the classification of [30]) which arise from ordinary LTSs by augmenting the outgoing transitions of a state s with probabilities that sum up to (at most) one. These models are based on Markov chains and replace the concept of non-deterministism by probabilistic branching. They are suitable e.g. for modelling the behaviour of probabilistic processes of synchronous calculi [29, 30, 45, 60, 6] or processes of a calculus with a probabilistic shuffle operator [2, 24]. For instance, the transition probabilities for the synchronous parallel composition $\mathcal{P}_1 \times \mathcal{P}_2$ of two sequential randomized processes $\mathcal{P}_1$ and $\mathcal{P}_2$ (each of them modelled by a fully probabilistic LTS) are obtained by multiplying the transition probabilities for the individual moves of $\mathcal{P}_1$ and $\mathcal{P}_2$. This relies on the assumption that that each step of $\mathcal{P}_1 \times \mathcal{P}_2$ is composed by the simultanous execution of transitions of both components $\mathcal{P}_1$ and $\mathcal{P}_2$ where the probabilistic choices are resolved independently. (For further details see e.g. [30, 6].)

On the other hand, there are several models that are based on Markov decision processes where *non-deterministism and probabilistic branching* coexist. Various variants of transition systems with non-deterministic and probabilistic choice are proposed and used as operational models for probabilistic processes with asynchronous parallelism in which case the non-determinism is used to describe the *interleaving* of the subprocesses [32, 61, 54, 36, 35, 57, 56, 11, 23, 9]. Moreover, as observed by several authors, e.g. [56, 41], the non-determinism can also be used to *represent underspecification* (which can be removed in further refinement steps) or *incomplete information* about the environment of the system (on which the resolution of the non-deterministic choices depends).

In this paper, we follow the approach of Segala & Lynch [57, 56] and deal with an extension of ordinary LTSs where in any state s there is a non-deterministic choice that selects one of the outgoing transitions. Any transition is augmented by an action label and a probabilistic choice (formalized by a distribution μ that specifies the frequencies to reach the possible successor states). As a special case of this model (that we call *probabilistic LTS*, abbrev. PLTS) we obtain the *reactive* probabilistic model à la Larsen & Skou [44] (see also [30]) where for any state s and action α, there is at most one outgoing transition labelled by α (while in PLTSs any state s might have two or more outgoing transitions with the same action label). The reactive view assumes that a given system "reacts" (in a probabilistic way) on the stimuli of the environment. For the choice which action is executed no probabilistic assumptions are made (as its resolution depends on the unpredicatable offerings by the environment) while, for any action α that is executable in a state s, probabilities are assigned to the possible successor states which might be reached when

executing α in state s. However, to model the interleaving behaviour of two processes e.g. of a *CCS*-like asynchronous parallel algebra (where the same action names might be used to specify the components $\mathcal{P}$ and $\mathcal{Q}$ of a parallel process $\mathcal{P} \| \mathcal{Q}$) the reactive view is not adequate. For example, the asynchronous parallel composition $\alpha.\mathcal{P}' \| \alpha.\mathcal{Q}'$ of two processes $\mathcal{P} = \alpha.\mathcal{P}'$ and $\mathcal{Q} = \alpha.\mathcal{Q}'$ yield a probabilistic LTS with two α-labelled outgoing transitions from the initial state representing the cases where $\mathcal{P}$ or $\mathcal{Q}$ performs the first step. This choice cannot be viewed as a reaction on the environment. Without any additional information, e.g. the relative speed of $\mathcal{P}$ and $\mathcal{Q}$ or the knowledge of a concrete scheduler that decides which process performs the next step, probabilistic assumptions about the resolution of this choice make little sense.

Most implementation relations on ordinary LTSs that have been proven to be very useful for the design and analysis of non-probabilistic parallel systems have been extended for the probabilistic setting. In the landmark paper [44], Larsen & Skou proposed an extension of bisimulation equivalence for reactive probabilistic LTSs. Bisimulation equivalence have been adapted for fully probabilistic systems [30, 42, 2, 45, 60] and several models with probabilistic choice and non-determinism [36, 35, 57]. In the beginning of the nineteenth, a lot of research has been done to investigate simulation-like and other relations for fully probabilistic systems; see e.g. [18, 42] for trace, failure and ready equivalences, [16, 17, 21, 63] for testing equivalences, [40] for a simulation relation and [8] for weak bisimulation equivalence. For most of the proposed equivalences, efficient algorithms have been developed [17, 39, 18, 8]. It is worth noting that even the decision problem for trace equivalence in fully probabilistic systems can be solved in polynomial time [39] while trace equivalence in (non-probabilistic) LTSs is PSPACE complete [43]. The situation is slightly different when dealing with a model for probabilistic systems with non-determinism. Even though several implementation relations have been defined and discussed under various aspects such as compositionality or axiomatization (see e.g. [35] for an axiomatization for bisimulation equivalence, [65, 41] for testing relations, [57, 58, 64] for various kinds of simulations and [58] for a trace preorder), research on algorithmic aspects of such relations is rare. To the best of our knowledge, the literature lacks for any complexity (or even decidability) result for trace-based relations on action-labelled probabilistic systems with non-determinism, e.g. trace equivalence defined as the kernel of the trace distribution preorder [58]. However, for any probabilistic model that subsumes ordinary LTSs (like probabilistic LTSs in our sense), any conservative extension of an equivalence or preorder for LTSs meets the lower bound complexity for LTSs; thus, trace equivalence for such models is PSPACE hard.

The contribution of this paper is the development of algorithms that compute the *bisimulation equivalence* classes and the *simulation preorder* (as introduced by [44, 57]) in a probabilistic LTS. (Throughout the paper we only consider finite systems where the state space and the number of transitions are finite.) Our method for deciding bisimilarity is a variant of the prominent *partitioning algorithm* à la [43, 51]; i.e. it performs a sequence of refinement steps that replace a given partition by a finer one, eventually resulting in the set of bisimulation equivalence classes. The time complexity of our algorithm is $\mathcal{O}(mn(\log m + \log n))$ where m is the number of transitions and n the number of states. In various applications, e.g. when the system arises from the interleaving of l "sequential"

probabilistic systems, we may suppose that for each state s there are at most l outgoing transitions. Hence, $m \leq l \cdot n$. If l is treated as a constant then we obtain the time complexity $\mathcal{O}(n^2 \log n)$ for deciding bisimilarity in probabilistic LTSs (which is closed to the best known time complexity $\mathcal{O}(m \log n)$ [51] for bisimulation in ordinary LTSs as $\mathcal{O}(m) = \mathcal{O}(n^2)$ in the worst case). The main idea for the computation of the simulation preorder is to reduce the question of whether a state "simulates" another one to a *maximum flow problem* in an appropriate network. Using the $\mathcal{O}(n^3/\log n)$ algorithm of [15] to determine the maximum flow, the algorithm runs in time $\mathcal{O}((mn^6 + m^2n^3)/\log n))$. The idea of using maximum flow problems is also applicable for the simulation-like relation à la Jonsson & Larsen [40] in fully probabilistic systems.

The remainder of the paper is organized as follows: In Section 2, we briefly explain some notations that we use in the further sections. Section 3 introduces the notions of a probabilistic LTS and recalls the definitions of bisimulations and simulations on them. Section 4 presents the algorithm for deciding bisimilarity, Section 5 the algorithm for computing the simulation preorder. In Section 6, we deal with fully probabilistic LTSs and briefly explain how the ideas of Section 5 can be used for computing the "satisfaction relation" à la [40]. Section 7 contains some concluding remarks.

We illustrate the key ideas by simple abstract examples without any concrete meaning. We refer the reader to e.g. [56, 55, 59] where examples are given that demonstrate how PLTSs can be used in realistic applications. Examples that give insides in the use of the fully probabilistic model in realistic situations can be found e.g. in [45, 37, 33, 34].

2 Preliminaries

We briefly explain some notations that are used throughout this paper. Further on, we recall some basic concepts concerning distributions, ordered balanced trees and networks.

Sets: For Z to be a set, 2^Z is the powerset of Z. If Z is finite then $|Z|$ denotes the number of elements of Z.

Equivalences and partitions: If R is an equivalence relation on a set Z then Z/R denotes the set of equivalence classes and, for $z \in Z$, $[z]_R$ the equivalence class of z w.r.t. R. A *partition* of a nonempty set Z is a set X consisting of pairwise disjoint nonempty subsets of Z such that $\bigcup_{B \in X} B = Z$. We often refer to the elements of a partition as *blocks*. If $z \in Z$ then $[z]_X$ denotes the unique block $B \in X$ that contains x, R_X the induced equivalence on Z that identifies exactly those elements z, $z' \in Z$ where $[z]_X = [z']_X$. A partition X is called *finer* than a partition X' (and X' is called *coarser* than X) iff each $B \in X$ is contained in some $B' \in X'$ (i.e. iff $R_{X'} \subseteq R_X$).

Distributions: A *distribution* on a finite set S is a function $\mu : S \to [0,1]$ such that

$$\sum_{s \in S} \mu(s) \;=\; 1.$$

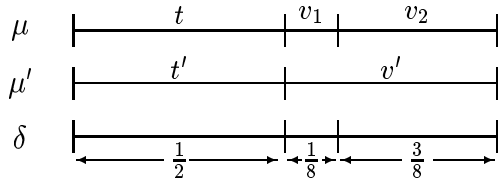
Let $Supp(\mu) = \{s \in S : \mu(s) > 0\}$ denote the *support* of μ. $Distr(S)$ denotes the set of distributions on S. We extend a distribution μ to a function $2^S \to [0,1]$ (also called

μ) which assigns to each subset U of S the probability for U, i.e. we define $\mu(U) = \sum_{s\in U} \mu(s)$. If $s \in S$ then μ_s^1 denotes the unique distribution on S with $\mu_s^1(s) = 1$ (and $\mu_s^1(s') = 0$ for all $s' \in S \setminus \{s\}$). If R is an equivalence on S then the induced equivalence $\equiv_R$ on $Distr(S)$ is given by $\mu \equiv_R \mu'$ iff $\mu(B) = \mu'(B)$ for any $B \in S/R$. If X is a partition on S and R on equivalence on S then $\equiv_X = \equiv_{R_X}$, i.e. $\mu \equiv_X \mu'$ iff $\mu(B) = \mu'(B)$ for all $B \in X$. Let $R \subseteq S \times S$ and μ, $\mu' \in Distr(S)$. A *weight function* for (μ, μ') w.r.t. R is a function $\delta : S \times S \to [0, 1]$ which satisfies:

1. If $\delta(s, s') > 0$ then $(s, s') \in R$.
2. For all s, $s' \in S$: $\sum_{s'\in S} \delta(s, s') = \mu(s)$, $\quad \sum_{s\in S} \delta(s, s') = \mu'(s')$

We write $\mu \sqsubseteq_R \mu'$ iff there exists a weight function for (μ, μ') w.r.t. R. It is easy to see that, if R is an equivalence then $\equiv_R$ and $\sqsubseteq_R$ coincide.

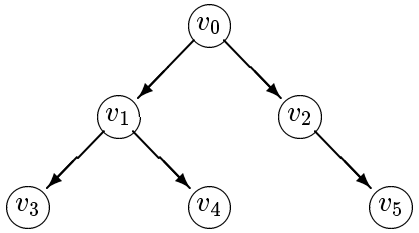
Example 2.1 Let $S = \{t, t', v_1, v_2, v'\}$ and R the equivalence on S where $S/R = \{T, V\}$ where $T = \{t, t'\}$, $V = \{v_1, v_2, v'\}$. Let μ, μ' be given by $\mu(t) = \mu'(t') = \mu'(v') = 1/2$, $\mu(v_1) = 1/8$ and $\mu(v_2) = 3/8$. Clearly, we have $\mu(T) = \mu'(T) = 1/2$ and $\mu(V) = \mu'(V) = 1/2$; thus, $\mu \equiv_R \mu'$. A weight function for (μ, μ') w.r.t. R can be obtained by combining fragments of equivalent elements according to the following figure.



Then, $\delta(t, t') = 1/2$, $\delta(v_1, v') = 1/8$, $\delta(v_2, v') = 3/8$ (and $\delta(x, y) = 0$ in all other cases) yields a weight function for (μ, μ') w.r.t. R. Thus, $\mu \sqsubseteq_R \mu'$. ■

Ordered balanced trees: For the implementation of the algorithms for deciding bisimilarity and similiarity, we propose the use of ordered balanced trees for the computation of certain equivalence classes. We briefly explain our notations. Let I be a nonempty and finite set and p_i, $i \in I$, real numbers. By an *ordered balanced tree for* p_i, $i \in I$, we mean a binary balanced tree (e.g. an AVL-tree [1] or a BB[α]-tree [49]) which arises by successively inserting the elements p_i, $i \in I$, (in any order) and performing the necessary rebalance steps. Each node v is labelled by a key-value $v.key \in \{p_i : i \in I\}$ such that $v_l.key < v.key < v_r.key$ for all nodes v_l (v_r) in the left (right) subtree of v. The construction of an ordered balanced tree for p_i, $i \in I$, takes $\mathcal{O}(|I| \log(r+1))$ time and $\mathcal{O}(|I|)$ space where r is the cardinality of $\{p_i : i \in I\}$. We also use additional labels for the nodes that can be derived from the set $v.indices = \{i \in I : p_i = v.key\}$. We describe the additional labels by their final value (i.e. the value in the final tree).

Example 2.2 Let $I = \{1, \ldots, 10\}$ and $p_1 = p_4 = 5$, $p_2 = p_7 = p_8 = 7$, $p_3 = 4$, $p_5 = 3$, $p_6 = 2$, $p_9 = p_{10} = 0$. The final tree depends on the type of ordered trees (e.g. AVL or BB[α]) and on the order in which the elements p_i are inserted. For instance, it is possible to obtain the following final tree.



$$\begin{array}{llllll} v_0.key & = & 4 & v_0.indices & = & \{3\} \\ v_1.key & = & 2 & v_1.indices & = & \{6\} \\ v_2.key & = & 5 & v_2.indices & = & \{1,4\} \\ v_3.key & = & 0 & v_3.indices & = & \{9,10\} \\ v_4.key & = & 3 & v_4.indices & = & \{5\} \\ v_5.key & = & 7 & v_5.indices & = & \{2,7,8\} \end{array}$$

If we deal with a function $l : S \to \{1, \ldots, 10\}$ where $S = \{s_1, s_2, s_3, s_4\}$ and $l(s_1) = l(s_2) = 5$, $l(s_3) = 6$, $l(s_4) = 7$ and the additional labels $v.states = \{s \in S : l(s) \in v.key\}$ then we have: $v_i.states = \emptyset$, $i = 0, 1, 3, 4$, $v_2.states = \{s_1, s_2\}$, $v_5.states = \{s_4\}$. ■

Networks: We briefly recall the basic definitions of networks. For further details see e.g. [26]. A *network* is a tuple $\mathcal{N} = (N, E, \bot, \top, c)$ where (N, E) is a finite directed graph (i.e. N is a set of nodes, $E \subseteq N \times N$ a set of edges) with two specified nodes $\bot$ (the *source*) and $\top$ (the *sink*) and a *capacity* c, i.e. a function c which assigns to each edge $(v, w) \in E$ a non-negative number $c(v, w)$. A *flow function* f for $\mathcal{N}$ is a function which assigns to edge e a real number $f(e)$ such that:

- $0 \leq f(e) \leq c(e)$ for all edges e.
- Let $in(v)$ be the set of incoming edges to node v and $out(v)$ the set of outgoing edges from node v. Then, for each node $v \in N \setminus \{\bot, \top\}$:

$$\sum_{e \in in(v)} f(e) \;=\; \sum_{e \in out(v)} f(e)$$

The *flow* $\mathcal{F}(f)$ of f is given by

$$\mathcal{F}(f) \;=\; \sum_{e \in out(\bot)} f(e) \;-\; \sum_{e \in in(\bot)} f(e).$$

The *maximum flow* in $\mathcal{N}$ is the supremum (maximum) over the flows $\mathcal{F}(f)$ where f is a flow function in $\mathcal{N}$. Algorithms to compute the maximum flow are given e.g. in [22, 28, 46, 15].

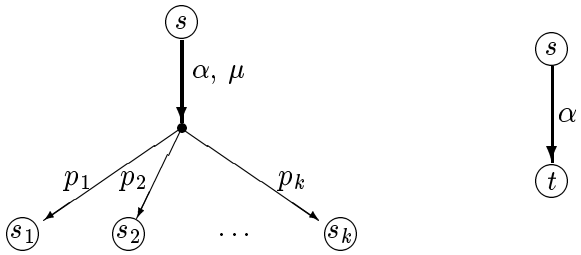
3 Probabilistic labelled transition systems

In this section we introduce our model (which essentially agrees with the *simple probabilistic automaton* à la Segala & Lynch [57]) and recall the definition of a bisimulation [44] and a simulation [57]. The reader should recall our notations that we use for distributions (see Section 2).

Definition 3.1 A *probabilistic labelled transition system* (PLTS) is a tuple $(S, Act, \to)$ where S is a finite set of states, Act a finite set of actions and $\to \subseteq\ S \times Act \times Distr(S)$ a finite set, called the transition relation. A *probabilistic process* is a tuple $\mathcal{P} = (S, Act, \to, s_{init})$ consisting of a PLTS $(S, Act, \to)$ and an initial state $s_{init} \in S$. ■

We refer to $s \xrightarrow{\alpha} \mu$ as a *transition* or a *step*. A state s is said to be *terminal* iff there is no outgoing transition from s. We often write $s \xrightarrow{\alpha} t$ instead of $s \xrightarrow{\alpha} \mu_t^1$. If $M \subseteq Distr(S)$ then $s \xrightarrow{\alpha} M$ denotes that $s \xrightarrow{\alpha} \mu$ for some $\mu \in M$. Similarly, if $C \subseteq S$ then $s \xrightarrow{\alpha} C$ stands short for $\exists t \in C : s \xrightarrow{\alpha} t$. Non-probabilistic LTSs (where the transition relation $\rightarrow$ is a subset of $S \times Act \times S$) arise as special cases of PLTSs by identifying each "non-probabilistic" transition $s \xrightarrow{\alpha} t$ with the "probabilistic" transition $s \xrightarrow{\alpha} \mu_t^1$.

We draw probabilistic transition system as follows. The states are in circles. A transition $s \xrightarrow{\alpha} \mu$ (with $Supp(\mu) = \{s_1, \ldots, s_k\}$ and $\mu(s_i) = p_i$) is depict as shown on the left of the following picture. Transitions of the form $s \xrightarrow{\alpha} t$ are also often depict as shown in the right.



For each state s, the outgoing transitions $s \xrightarrow{\alpha} \mu$ represent the non-deterministic alternatives in the state s. The non-deterministic choices are beyond the control of the process and is supposed to be resolved by the environment (formalized by a "scheduler" [32, 61] or an "adversary" [57, 56]), whereas the probabilistic choices are made by the system itself, according to the underlying distribution μ. Given a state s, the environment chooses some outgoing transition $s \xrightarrow{\alpha} \mu$. Then, the action α is performed and the system resolves the probabilistic choice, i.e. with probability $\mu(t)$ the state t is reached afterwards.

Reactive PLTSs [44, 30] are the probabilistic counterpart to *deterministic* LTS where for any state and action α at most one outgoing transition from s is labelled by α. Reactive PLTSs rely on the assumption that the environment determines which actions are possible. For any of these actions, probabilities specify the frequencies of the possible reactions of the system.

Definition 3.2 (cf. [44, 30]) A PLTS $(S, Act, \rightarrow)$ is called *reactive* iff for each $s \in S$ and $\alpha \in Act$: if $s \xrightarrow{\alpha} \mu$ and $s \xrightarrow{\alpha} \mu'$ then $\mu = \mu'$. ■

We now recall the definition of bisimulations and simulations in PLTSs. The reader should recall the notion of a weight function and the definitions of the relations $\equiv_R$ and $\sqsubseteq_R$ that we presented in Section 2. Bisimulation equivalence $\sim$ and the simulation preorder $\sqsubseteq$ are defined as relations on the states of a (single) systems (see Definitions 3.3 and 3.6). These definitions can easily be adapted for probabilistic processes with the same action set Act. Given two probabilistic processes $\mathcal{P} = (S, Act, \rightarrow, s_{init})$ and $\mathcal{P}' = (S', Act, \rightarrow, s'_{init})$ we define $\mathcal{P} \sim \mathcal{P}'$ iff $s_{init} \sim s'_{init}$ where s_{init} and s'_{init} are viewed as states in the composed system that arsise by taking the disjoint union of the state spaces S and S'. In a similar way, we adapt the simulation preorder $\sqsubseteq$ to probabilistic processes.

Bisimulation: Given a (non-probabilistic) LTS $(S, Act, \rightarrow)$, the standard definition of bisimulation equivalence [47, 52] can be reformulated as the coarsest equivalence R on the state space S such that for all $(s, s') \in R$ and all transitions $s \xrightarrow{\alpha} t$ there is a transition $s' \xrightarrow{\alpha} t'$ with $(t, t') \in R$. In the probabilistic setting, we take quantitative aspects into

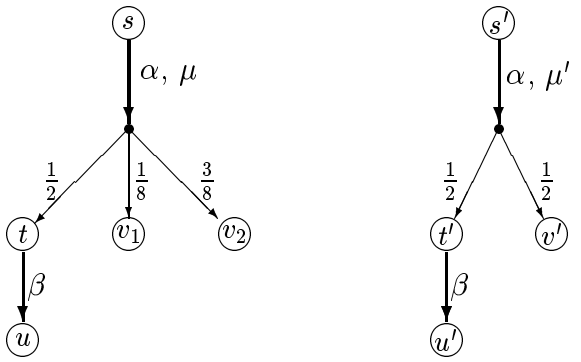


Figure 1: $s \sim s'$

account and require that for bisimilar states s, s' and any transition $s \xrightarrow{\alpha} \mu$ from s there is a "matching" transition $s' \xrightarrow{\alpha} \mu'$ from s' where "matching" means that the probabilities for the bisimulation equivalence classes with respect to μ and μ' are the same.

Definition 3.3 (cf. [44, 57]) A *bisimulation* on a PLTS $(S, Act, \rightarrow)$ is an equivalence relation R on S such that for all $(s, s') \in R$: If $s \xrightarrow{\alpha} \mu$ then there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu \equiv_R \mu'$. Two states s and s' are called *bisimilar* (denoted by $s \sim s'$) iff there exists a bisimulation which contains (s, s'). ■

An alternative description of bisimulation is based on weight functions for distributions and uses the fact that $\equiv_R$ agrees with $\sqsubseteq_R$ if R is an equivalence. In essential, this result goes back to Jonsson & Larsen (Theorem 4.6 in [40]) who deal with a fully probabilistic model with labels for the states rather than action labels. A similar observation for reactive PLTSs was made by de Vink & Rutten [62] who use a categorical characterization of what we call weight functions.

Lemma 3.4 *Let $(S, Act, \rightarrow)$ be a PLTS and R an equivalence relation on S. Then, R is a bisimulation iff for all $(s, s') \in R$: if $s \xrightarrow{\alpha} \mu$ then there exists $s' \xrightarrow{\alpha} \mu'$ with $\mu \sqsubseteq_R \mu'$.*

Example 3.5 The states s and s' in the system shown in Figure 1 are bisimilar. More precisely, $S/\sim = \{\{s, s'\}, T, V\}$ where $T = \{t, t'\}$ and $V = \{u, u', v_1, v_2, v'\}$. The transition $s \xrightarrow{\alpha} \mu$ is "matched" by $s' \xrightarrow{\alpha} \mu'$ (in the sense that $\mu \equiv_\sim \mu'$) as we have

$$\mu(\{s, s'\}) = 0 = \mu'(\{s, s'\})$$
$$\mu(T) = \mu(t) = \tfrac{1}{2} = \mu'(t') = \mu'(T)$$
$$\mu(V) = \mu(v_1) + \mu(v_2) = \tfrac{1}{8} + \tfrac{3}{8} = \tfrac{1}{2} = \mu'(v') = \mu'(V).$$

A weight function for (μ, μ') w.r.t. $\sim$ was given in Example 2.1. (Thus, $\mu \sqsubseteq_\sim \mu'$.) ■

Simulation: As in the non-probabilistic case, simulation can be viewed as "uni-directional bisimulation" in the sense that a state s' "simulates" another state s if each step of s can be "simulated" by a step of s'. (In that case, the process $\mathcal{P}$ with initial state s can be viewed as an "implementation" of the process $\mathcal{P}'$ with initial state s' as each step of $\mathcal{P}$ is "allowed" by the "specification" $\mathcal{P}'$.) For (non-probabilistic) LTSs, the formal definition of a simulation is obtained from the notion of a bisimulation by dropping the symmetry.

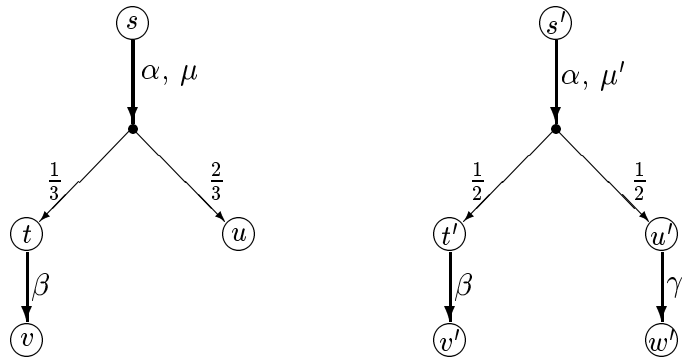
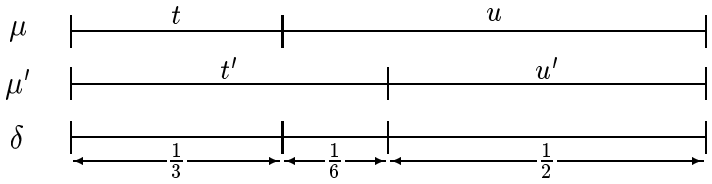


Figure 2: $s \sqsubseteq s'$

That is, a simulation on a LTS is a binary relation R on S such that $(s, s') \in R$ and $s \xrightarrow{\alpha} t$ implies $s' \xrightarrow{\alpha} t'$ for some state t' with $(t, t') \in R$. (See [48].) To obtain the formal definition of a simulation in the probabilistic setting, we take as basis the characterization of a bisimulation given in Lemma 3.4 and skip the requirement R to be an equivalence.

Definition 3.6 (cf. [57]) A *simulation* for a PLTS $(S, Act, \rightarrow)$ is a subset R of $S \times S$ such that for all $(s, s') \in R$: If $s \xrightarrow{\alpha} \mu$ then there exists a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu \sqsubseteq_R \mu'$. We say s *implements* s' (and s' *simulates* s), denoted by $s \sqsubseteq s'$, iff there exists a simulation which contains (s, s'). s, s' are called *similar* (written $s \sim_{\text{sim}} s'$) iff $s \sqsubseteq s'$ and $s' \sqsubseteq s$. ■

Example 3.7 Consider the PLTS of Figure 2. Clearly, $u \sqsubseteq u'$ and $u, t \sqsubseteq t'$. A weight function for (μ, μ') can be obtained by combining certain parts of t (of u) with certain parts of t' (of u' and t'). The weight function δ for (μ, μ') w.r.t. $\sqsubseteq$ is given by: $\delta(t, t') = 1/3$, $\delta(u, t') = 1/6$, $\delta(u, u') = 1/2$.



We obtain $s \sqsubseteq s'$. ■

Remark 3.8 It is easy to see that the above notions of bisimulations and simulations applied to a non-probabilistic LTS agree with the standard notions of (bi-)simulations à la [47, 52, 48]. For this, we just have to use the simple fact that $\mu_t^1 \sqsubseteq_R \mu_{t'}^1$ iff $(t, t') \in R$. (In particular, if R is an equivalence then $\mu_t^1 \equiv_R \mu_{t'}^1$ iff $(t, t') \in R$.) ■

It is clear that $\sqsubseteq$ is a preorder whose kernel $\sim_{\text{sim}} = \sqsubseteq \cap \sqsubseteq^{-1}$ is coarser than bisimulation equivalence, i.e. $s \sim s'$ implies $s \sim_{\text{sim}} s'$. Similarly to the non-probabilistic case, in general, $\sim_{\text{sim}}$ does not coincide with bisimulation, while $\sim = \sim_{\text{sim}}$ for reactive PLTSs [9, 5]. The latter observation can be viewed as the probabilistic counterpart to the well-known result stating that bisimulation and simulation equivalence coincide for deterministic LTSs.

Remark 3.9 One might expect that the definition of bisimulations on PLTSs (Definition 3.3) yields simpler possibilities to drop the symmetry; thus yielding alternative definitions of a simulation preorder that do not use (the quite complex concept of) weight functions. One possibility is to consider the downward closure $t\downarrow_R = \{u \in S : (u,t) \in R\}$ of all elements $t \in S$ (rather than the equivalence classes $[t]_R$) and to define $\mu \sqsubseteq_R \mu'$ iff $\mu(t\downarrow_R) \geq \mu'(t\downarrow_R)$ for all $t \in S$. Another possibility is to deal with the upward closures $t\uparrow_R = \{u \in S : (u,t) \in R\}$. Both possibilities yield a preorder that is strict coarser than the simulation preorder à la [57]. We argue that none of these relations can be viewed as a probabilistic counterpart to Milner's simulation preorder.

We define a $\downarrow$-*simulation* on a PLTS $(S, Act, \rightarrow)$ to be a binary relation R on S such that for all $(s,s') \in R$ and $s \xrightarrow{\alpha} \mu$ there exists $s' \xrightarrow{\alpha} \mu'$ with $\mu(t\downarrow_R) \geq \mu'(t\downarrow_R)$ for all $t \in S$. Similarly, a $\uparrow$-*simulation* is a binary relation R on S such that for all $(s,s') \in R$ and $s \xrightarrow{\alpha} \mu$ there exists $s' \xrightarrow{\alpha} \mu'$ with $\mu(t\uparrow_R) \geq \mu'(t\uparrow_R)$ for all $t \in S$. We define $s \sqsubseteq_\downarrow s'$ ($s \sqsubseteq_\uparrow s'$) iff $(s,s') \in R$ for some $\downarrow$-simulation ($\uparrow$-simulation). Using the results of [9], we obtain that the simulation preorder $\sqsubseteq$ is a $\downarrow$-simulation and a $\uparrow$-simulation. Thus, $s \sqsubseteq s'$ implies $s \sqsubseteq_\downarrow s'$ and $s \sqsubseteq_\uparrow s'$.

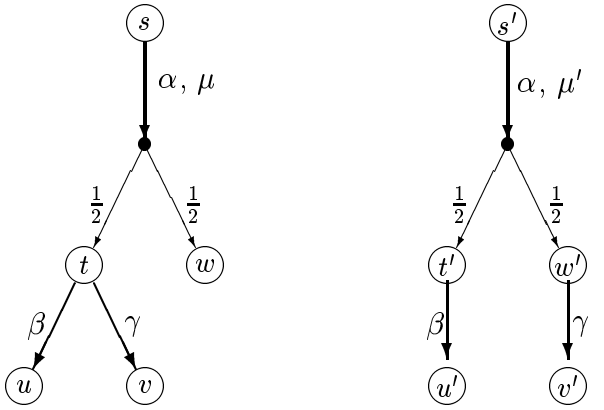


Figure 3: $s \sqsubseteq_\downarrow s'$ and $s \not\sqsubseteq s'$

In the PLTS of Figure 3 we have $s \sqsubseteq_\downarrow s'$ but $s \not\sqsubseteq s'$. From the branching time view, the process on left (i.e. the process with initial state s) should not be considered to be an implemention of the process on the right (i.e. the process with initial state s') as s can reach a state where both actions γ and β can be performed while s' cannot. In the PLTS of Figure 4, we have $s \sqsubseteq_\uparrow s'$ but $s \not\sqsubseteq s'$. For our opinion, the process on the right (the process with initial state s') cannot be viewed as a simulation of the process on the left (the process with initial state s) since s reachs a non-terminal state after performing α with probability 1, while s' can reach a terminal state (w') after performing α with non-zero probability. Even the relation $\sqsubseteq_\uparrow \cap \sqsubseteq_\downarrow$ is coarser than $\sqsubseteq$. In the system of Figure 4, we add a transition $w' \xrightarrow{\alpha} w'$ and obtain $s \not\sqsubseteq s'$ while $(s \sqsubseteq_\uparrow s') \wedge (s \sqsubseteq_\downarrow s')$. ■

4 Computing the bisimulation equivalence classes

In this section, we present a polynomial time algorithm for computing the quotient space of a PLTS w.r.t. bisimulation equivalence $\sim$.

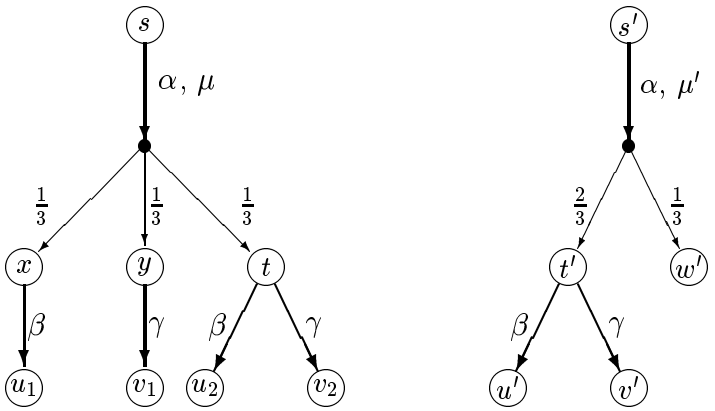


Figure 4: $s \sqsubseteq_{\uparrow} s'$ and $s \not\sqsubseteq s'$

The general schema: The basic idea for our algorithm is the use of the prominent *partitioning technique* [43, 51] for the (non-probabilistic) LTSs (see Figure 5). We start with the trivial partition $X = \{S\}$ and then successively refine X by splitting any block B of X into subblocks, finally resulting in the bisimulation equivalence classes. (The notion of a partition and related notations were explained in Section 2.) In [43, 51] the

$X := \{S\}$;
Repeat
    $X_{old} := X$;
    $X := \mathit{Refine}(X)$;
until $X_{old} = X$;
Return X.

Figure 5: Schema for computing the bisimulation equivalence classes

refinement operator $\mathit{Refine}(X)$ depends on a *splitter* for X, i.e. a pair $\langle \alpha, C \rangle$, $\alpha \in \mathit{Act}$, $C \in X$, such that, for some $B \in X$, the sets $B_{(\alpha,C)} = \{s \in B : s \xrightarrow{\alpha} C\}$ and $B \setminus B_{(\alpha,C)}$ are nonempty. (See Figure 6.) For $\langle \alpha, C \rangle$ to be a splitter candidate for X (i.e. an action/block pair), the refinement operator $\mathit{Refine}(X) = \mathit{Refine}(X, \alpha, C)$ yields the partition

$$\mathit{Refine}(X, \alpha, C) \;=\; \bigcup_{B \in X} \mathit{Refine}(B, \alpha, C)$$

where $\mathit{Refine}(B, \alpha, C) \;=\; \{B_{(\alpha,C)}, B \setminus B_{(\alpha,C)}\} \setminus \{\emptyset\}$. Using several additional tricks, this method can be implemented in time $\mathcal{O}(m \log n)$ [51] (where $n = |S|$ is the number of states and $m = |\rightarrow|$ the number of transitions) and modified for fully probabilistic systems and reactive PLTSs (with roughly the same complexity) [39]. (See Section 6 for the definition of fully probabilistic systems and bisimulation on them.) However, even though it fails for (general) PLTSs when we deal with action/block pairs as splitters (see Example 4.2), a suitable modification of the partitioning algorithm works for PLTSs.

Notation 4.1 In the sequel, we fix a PLTS $(S, \mathit{Act}, \rightarrow)$. Recall that (according to Definition 3.1), all components are finite, i.e. the state space S, the action set Act

and the transition relation $\rightarrow$ are finite. For $s \in S$, let $\mathit{Steps}(s) = \{(\alpha, \mu) : s \stackrel{\alpha}{\rightarrow} \mu\}$, $\mathit{Steps}_\alpha(s) = \{\mu : s \stackrel{\alpha}{\rightarrow} \mu\}$ and $\mathit{act}(s) = \{\alpha \in \mathit{Act} : \mathit{Steps}_\alpha(s) \neq \emptyset\}$. Let $m_s = |\mathit{Steps}(s)|$ (the number of outgoing transitions from s), $m_{\alpha,s} = |\mathit{Steps}_\alpha(s)|$ (the number of α-labelled outgoing transitions from s), $n = |S|$ (the number of states) and $m = \sum_{s \in S} m_s$ (the total number of transitions). When we analyze the complexity we assume that there is no irrelevant action $\alpha \in \mathit{Act}$. Formally, we make the additional requirement that any action $\alpha \in \mathit{Act}$ occurs as label of some transition, i.e. $\mathit{Steps}_\alpha(s) \neq \emptyset$ for at least one state s. (Thus, $|\mathit{Act}| \leq m$.) ■

Before we present our partitiong algorithm for PLTSs, we briefly sketch how the above mentioned splitter/partitioning technique can be modified for reactive PLTSs and explain why this method fails for general PLTSs.

The splitter/partitioning technique for reactive PLTSs: We briefly recall the method suggested by Huynh & Tian [39] which was originally formulated for fully probabilistic systems but – as observed in [39] – is also applicable for reactive PLTSs. Let $(S, \mathit{Act}, \rightarrow)$ be a reactive PLTS. If $C \subseteq S$ then we write $\mathbf{P}(s, \alpha, C)$ to denote the probability to move from state s via the action α to a state of C. (Formally, $\mathbf{P}(s, \alpha, C) = \mu(C)$ if $s \stackrel{\alpha}{\rightarrow} \mu$. For $\alpha \notin \mathit{act}(s)$ we put $\mathbf{P}(s, \alpha, C) = 0$.) Given an action/block pair $\langle \alpha, C \rangle$ and a partition X, the refinement operator $\mathit{Refine}(X, \alpha, C)$ splits any block $B \in X$ into the equivalence classes w.r.t. the equivalence $\approx_{(a,C)}$ where $s \approx_{(a,C)} s'$ iff $\mathbf{P}(s, \alpha, C) = \mathbf{P}(s', \alpha, C)$, i.e.

$$\mathit{Refine}(X, \alpha, C) \;=\; \bigcup_{B \in X} \mathit{Refine}(B, \alpha, C) \quad \text{where} \quad \mathit{Refine}(B, \alpha, C) = B/\approx_{(\alpha,C)} .$$

An implementation of the algorithm sketched in Figure 6 might use a queue Q that

```
X := {S};
While there exists a splitter ⟨α, C⟩ of X do
        X := Refine(X, α, C);
Return X.
```

Figure 6: Partitioning/splitter algorithm in finite LTSs or reactive PLTSs

organizes the possible splitters. Initially, Q contains the pairs $\langle \alpha, S \rangle$, $\alpha \in \mathit{Act}$. As long as Q is nonempty, we take the first element $\langle \alpha, C \rangle$ of Q, remove $\langle \alpha, C \rangle$ from Q and refine X according to $\langle \alpha, C \rangle$ as follows. For each $B \in X$, we compute the probabilities

$$p_s \;=\; \begin{cases} 0 & : \text{ if } \mathit{Steps}_\alpha(s) = \emptyset \\ \mu(C) & : \text{ if } \mathit{Steps}_\alpha(s) = \{\mu\} \end{cases} , \quad s \in B.$$

We construct an ordered balanced tree $T_{(B,\alpha,C)}$ for p_s, $s \in B$, with additional labels $v.\mathit{states}$ for each node v such that finally $v.\mathit{states} = \{s \in B : v.\mathit{key} = p_s\}$. (The reader should recall our notations explained in Section 2 that we use for ordered balanced trees.) The nodes in the final tree represent $\mathit{Refine}(B, \alpha, C)$, more precisely

$$\mathit{Refine}(B, \alpha, C) \;=\; \{v.\mathit{states} : v \text{ is a node in } T_{(B,\alpha,C)}\}.$$

If $\mathit{Refine}(B, \alpha, C) \neq \{B\}$ then for each $B' \in \mathit{Refine}(B, \alpha, C)$ but one of the largest we add the pairs $\langle \beta, B' \rangle$, $\beta \in \mathit{Act}$, to the end of Q. (By the largest blocks we mean those blocks $B' \in \mathit{Refine}(B, \alpha, C)$ where $|B'|$ is maximal.) An implementation of these ideas which uses similar tricks as suggested in [51] for the non-probabilistic case (see also [27]) yields the following complexity result.

Theorem 1 *The bisimulation equivalence classes of a reactive PLTS (with n states and m transitions) can be computed in time $\mathcal{O}(mn \log n)$ and space $\mathcal{O}(mn)$.*

As bisimulation and simulation equivalence coincide for reactive PLTSs [9, 5], we get the following theorem as a corollary of Theorem 1.

Theorem 2 *The simulation equivalence classes of a reactive PLTS (with n states and m transitions) can be computed in time $\mathcal{O}(mn \log n)$ and space $\mathcal{O}(mn)$.*

The following example explains why the splitter technique fails for general PLTSs when we deal with action/block pairs as splitters and the refinement operator $\mathit{Refine}(X, \alpha, C) = \bigcup_{B \in X} B/\approx_{(\alpha,C)}$ where $s \approx_{(\alpha,C)} s'$ iff for any transition $s \xrightarrow{\alpha} \mu$ there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu(C) = \mu'(C)$ (and vice versa).

Example 4.2 We consider a PLTS as shown in Figure 7 where we suppose that $t \sim t'$, $u \sim u'$, $v \sim v'$, $w \sim w'$ and that t, u, v, w are pairwise non-bisimilar. (The outgoing transitions of the states $t, t', u, u', v, v', w, w'$ are omitted in the picture.) Then, $s \not\sim s'$ as there is no transition from s' that matches the transition $s \xrightarrow{\alpha} \nu$ where $\nu(t) = \nu(u) = 1/2$. On the other hand, s, s' cannot be distinguished by action/block pairs. More precisely, for any block C of a partition X that is coarser than the quotient space $S/\sim$ w.r.t. bisimulation equivalence, and for any transition $s \xrightarrow{\alpha} \mu$ there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu(C) = \mu'(C)$. Thus, the splitter/partitioning algorithm sketched in Figure 6 would return that s and s' are bisimilar. ■

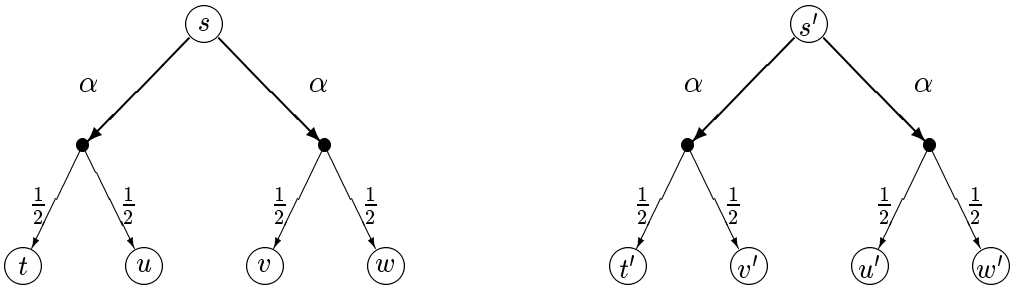


Figure 7: $s \not\sim s'$, but s and s' cannot be distinguished by action/block pairs

The partitioning algorithm for PLTSs: We now explain how the bisimulation equivalence classes can be computed in general PLTSs. We maintain the schema sketched at the beginning of this section (Figure 5) but work with two partitions: a partition X for the states and a partition $\mathcal{M}$ for the steps.

Notation 4.3 In what follows, by a *step partition* we mean a set $\mathcal{M}$ consisting of pairs $\langle \alpha, M \rangle$ where $\alpha \in \mathit{Act}$ and $M \subseteq \bigcup_{s \in S} \mathit{Steps}_\alpha(s)$ such that, for any action α, the set

$\{M : \langle \alpha, M\rangle \in \mathcal{M}\}$ is a partition of $\bigcup_{s\in S}\ \mathit{Steps}_\alpha(s)$. We refer to the elements of a step partition as *step classes*. Given two step partitions $\mathcal{M}$ and $\mathcal{M}'$, we say $\mathcal{M}$ is coarser than $\mathcal{M}'$ (and $\mathcal{M}'$ finer than $\mathcal{M}$) iff for any $\langle \alpha, M'\rangle \in \mathcal{M}'$ there is some step class $\langle \alpha, M\rangle \in \mathcal{M}$ with $M' \subseteq M$. Given a partition X for the state space S, the induced step partition $\mathcal{M}_X$ is given by the step classes $\langle \alpha, M\rangle$ where $M \in (\bigcup_{s\in S}\ \mathit{Steps}_\alpha(s))\,/\,\equiv_X$. (I.e. for any step classes $\langle \alpha, M\rangle$ in $\mathcal{M}_X$, the set M is a maximal (nonempty) subset of $\bigcup_{s\in S}\ \mathit{Steps}_\alpha(s)$ such that $\mu \equiv_X \mu'$ for all $\mu,\ \mu' \in M$.) ■

The basic idea of our algorithm (sketched in Figure 8) is that the step partition $\mathcal{M}$ yields an "overapproximation" for $\mathcal{M}_X$ (that is, $\mathcal{M}$ is coarser than $\mathcal{M}_X$); thus, it contains the information for which distributions $\mu,\ \mu' \in \bigcup_{s\in S}\ \mathit{Steps}_\alpha(s)$, $\mu \not\equiv_X \mu'$ is already detected. (Note that, if $\mu \in M$, $\mu' \in M'$ where $M \neq M'$ and $\langle \alpha, M\rangle,\ \langle \alpha, M'\rangle \in \mathcal{M}$ then $\mu \not\equiv_X \mu'$.) Our algorithm works as follows. We skip the first refinement step and start with the state partition

$$X_{init} \ = \ S/\approx_{Act} \quad \text{where } s \approx_{Act} s' \text{ iff } \mathit{act}(s) = \mathit{act}(s')$$

that identifies exactly those states that can perform the same actions. The initial step partition

$$\mathcal{M}_{init} \ = \ \{\langle \alpha, M_\alpha\rangle : \alpha \in \mathit{Act}\} \ \text{ where } \ M_\alpha = \bigcup_{s\in S}\ \mathit{Steps}_\alpha(s)$$

identifies all transitions with the same action label. In each iteration, we tro to refine the state partition X according to a step class $\langle \alpha, M\rangle$ of $\mathcal{M}$ or the step partition $\mathcal{M}$ according to a block $C \in X$. Splitting of $\mathcal{M}$ according to a block $C \in X$ (operation $\mathit{Split}(\mathcal{M}, C)$) means the replacement of any step class $\langle \alpha, M\rangle$ of $\mathcal{M}$ by the the step classes $\langle \alpha, M_1\rangle, \ldots, \langle \alpha, M_r\rangle$ where $M/\equiv_C \ = \ \{M_1, \ldots, M_r\}$ and $\mu \equiv_C \mu'$ iff $\mu(C) = \mu'(C)$. Formally,

$$\mathit{Split}(\mathcal{M}, C) \ = \ \bigcup_{\langle \alpha, M\rangle \in \mathcal{M}} \mathit{Split}(\langle \alpha, M\rangle, C)$$

where

$$\mathit{Split}(\langle \alpha, M\rangle, C) \ = \ \{\langle \alpha, M'\rangle : M' \in M/\equiv_C\}.$$

In the operation $\mathit{Refine}(X, \alpha, M)$, the step class $\langle \alpha, M\rangle \in \mathcal{M}$ serves as splitter and devides any block B of X into the subblocks $B_{(\alpha,M)} = \{s \in B : s \xrightarrow{\alpha} M\}$ and $B \setminus B_{(\alpha,M)}$. I.e.

$$\mathit{Refine}(X, \alpha, M) \ = \ \bigcup_{B\in X} \mathit{Refine}(B, \alpha, M)$$

where $\mathit{Refine}(B, \alpha, M) = \{B_{(\alpha,M)}, B \setminus B_{(\alpha,M)}\} \setminus \{\emptyset\}$ and $B_{(\alpha,M)} = \{s \in B : s \xrightarrow{\alpha} M\}$. If no further refinements of X and $\mathcal{M}$ are possible then $X = S/\sim$. This observation is formalized in the following lemma which yields the total correctness of our method (the algorithm sketched in Figure 8).

Lemma 4.4 *Let X be a partition of S which is coarser than $S/\sim$ and $\mathcal{M}$ a step partition where $\mathcal{M}$ is coarser than $\mathcal{M}_X$. Then, we have:*

(a) X_{init} is coarser than $S/\sim$ and $\mathcal{M}_{init}$ coarser than $\mathcal{M}_{X_{init}}$.

(b) For any $C \in X$, the step partition $\mathit{Split}(\mathcal{M}, C)$ is finer than $\mathcal{M}$ and coarser than $\mathcal{M}_X$. If $\mathcal{M} \neq \mathcal{M}_X$ then there is some block $C \in X$ where $\mathit{Split}(\mathcal{M}, C) \neq \mathcal{M}$.

$X := S/\approx_{Act}$ where $s \approx_{Act} s'$ iff $act(s) = act(s')$;
$\mathcal{M} := \{\langle \alpha, M_\alpha \rangle : \alpha \in Act\}$ where $M_\alpha = \bigcup_{s \in S} \ Steps_\alpha(s)$;
As long as X or $\mathcal{M}$ can be modified perform one of the following steps:

- either choose some $C \in X$ and put $\mathcal{M} := Split(\mathcal{M}, C)$
- or choose some step class $\langle \alpha, M \rangle \in M$ and put $X := Refine(X, \alpha, M)$;

Return X.

Figure 8: Basic algorithm for computing the bisimulation equivalence classes in PLTSs

(c) For any $\langle \alpha, M \rangle \in \mathcal{M}$, $Refine(X, \alpha, M)$ is a partition which is coarser than $S/\sim$. If $X \neq S/\sim$ then

- *either $\mathcal{M} \neq \mathcal{M}_X$ (and there is some $C \in X$ with $Split(\mathcal{M}, C) \neq \mathcal{M}$)*
- *or there is some $\langle \alpha, M \rangle \in \mathcal{M}$ with $Refine(X, \alpha, M) \neq X$.*

In particular, if $Refine(X, \alpha, M) = X$ for all $\langle \alpha, M \rangle \in \mathcal{M}$ and $Split(\mathcal{M}, C) = \mathcal{M}$ for all $C \in X$ then $X = S/\sim$ and $\mathcal{M} = \mathcal{M}_X$.

Proof: easy verification. ■

The two-phased partitioning technique: We now describe some details that yield the desired complexity and remove the non-determinism (the choice whether $\mathcal{M}$ or X will be refined). First, we make some simple observations that we use to avoid irrelevant refinement attempts. Clearly, it suffices to use any block C at most once for the operation $Split(\mathcal{M}, C)$. Moreover, if a block B has been devided into the proper subblocks $B_{(\alpha,M)}$ and $B \setminus B_{(\alpha,M)}$ then the operations $Split(\mathcal{M}, B_{(\alpha,M)})$ and $Split(\mathcal{M}, B \setminus B_{(\alpha,M)})$ have the same effect. Thus, it suffices to use only one of the subblocks $B_{(\alpha,M)}$ or $B \setminus B_{(\alpha,M)}$ as arguments for the operator $Split(\mathcal{M}, \cdot)$. This motivates the use of a set *NewBlocks* that keeps book about the blocks C for which the operation $Split(\mathcal{M}, C)$ still has to be performed.

- Initially, *NewBlocks* contains all blocks of X_{init} but one of the largest.
- A block C is removed from *NewBlocks* just when $Split(\mathcal{M}, C)$ is executed.
- Whenever the refinement operator $Refine(B, \alpha, M)$ returns a proper splitting of B into $B_{(\alpha,M)}$ and $B \setminus B_{(\alpha,M)}$ then we add the smaller of these two new blocks to *NewBlocks* (but ignore the other one).

Similarly, we use a set *NewStepClasses* that contains all step classes $\langle \alpha, M \rangle$ for which the operator $Refine(X, \alpha, M)$ might yield a new state partition.

- Initially, *NewStepClasses* contains all step classes in $\mathcal{M}_{init}$
- As soon as $Refine(X, \alpha, M)$ is executed, $\langle \alpha, M \rangle$ is removed from *NewStepClasses*.
- If the operator $Split(\mathcal{M}, C)$ returns a proper splitting of $\langle \alpha, M \rangle$ into two or more step classes $\langle \alpha, M_1 \rangle, \ldots, \langle \alpha, M_r \rangle$ then we add the new step classes $\langle \alpha, M_1 \rangle, \ldots, \langle \alpha, M_r \rangle$ to *NewStepClasses*.

Initialization:

$X := X_{init}$; $\mathcal{M} := \mathcal{M}_{init}$; $NewStepClasses := \mathcal{M}$;

$NewBlocks := X \setminus \{C\}$ where C is one of the largest blocks in X;

Iteration: Repeat

Phase 1: Splitting of $\mathcal{M}$ via $Split(\mathcal{M}, C)$

If $NewBlocks \neq \emptyset$ then

- choose some $C \in NewBlocks$ and remove C from $NewBlocks$;
- $\mathcal{M}_{old} := \mathcal{M}$; $\mathcal{M} := \emptyset$;
- For all $\langle \alpha, M \rangle \in \mathcal{M}_{old}$ do
  - Compute an enumeration $M_1, \ldots, M_r$ of $M/\equiv_C$;
  - $\mathcal{M} := \mathcal{M} \cup \{\langle \alpha, M_i \rangle : i = 1, \ldots, r\}$;
  - If $r \geq 2$ then insert $\langle \alpha, M_i \rangle$, $i = 1, \ldots, r$, into $NewStepClasses$;

Phase 2: Refinement of X via $Refine(X, \alpha, M)$

If $NewStepClasses \neq \emptyset$ then

- choose some step class $\langle \alpha, M \rangle \in NewStepClasses$;
- $NewStepClasses := NewStepClasses \setminus \{\langle \alpha, M \rangle\}$;
- $X_{old} := \mathcal{M}$; $X := \emptyset$;
- For all blocks $B \in X_{old}$ do
  - $B' := B_{(\alpha,M)}$; $B'' := B \setminus B_{(\alpha,M)}$; $X := X \cup (\{B', B''\} \setminus \{\emptyset\})$;
  - If $B' \neq \emptyset$ and $B'' \neq \emptyset$ then
    if $|B'| \leq |B''|$ then $C := B'$ else $C := B''$;
    $NewBlocks := NewBlocks \cup \{C\}$;

until $NewBlocks = \emptyset$ and $NewStepClasses = \emptyset$;

Output: Return X.

Figure 9: The two-phased partitioning algorithm

This leads to the algorithm shown in Figure 9. In each iteration step, we work with a *two-phased partitioning technique*. In the first phase, we refine the step partition $\mathcal{M}$ according to some $C \in NewBlocks$ (via the operator $Split(\mathcal{M}, C)$). This means the splitting of any step class $\langle \alpha, M \rangle \in \mathcal{M}$ into the step classes $\langle \alpha, M_1 \rangle, \ldots, \langle \alpha, M_r \rangle$ that we obtain when we apply the operator $Split(\langle \alpha, M \rangle, C)$. In the second phase, we refine the current state partition X according to some $\langle \alpha, M \rangle \in NewStepClasses$ (i.e. we replace X by the state partition obtained by the operation $Refine(X, \alpha, M)$).

With the use of the auxiliary sets of blocks and step classes (the sets *NewBlocks* and *NewStepClasses*) we get the following bounds on the total number of refinement operations $Refine(X, \alpha, M)$ and splitting operations $Split(\mathcal{M}, C)$.

Lemma 4.5 *The total number of refinement operations $Refine(X, \alpha, M)$ that are executed in phase 2 of the two-phased partitioning algorithm in Figure 9 is bounded by $2(m - 1)$.*

Proof: First we observe that for any finite set Z with $|Z| = k$ and any sequence $P_0, P_1, \ldots$ of partitions of Z such that P_j is finer than P_{j-1} we have $|P_0 \cup P_1 \cup \ldots| \leq 2(|Z| - 1)$. (We omit the proof of this simple fact which follows by induction on $|Z|$.)

Let $\mathcal{M}_0 = \mathcal{M}_{init}$ and $\mathcal{M}_j$ the step partition $\mathcal{M}$ at the end of the j-th iteration. The total number J of refinement steps $\mathit{Refine}(X, \alpha, M)$ agrees with the number of step classes that are once inserted into *NewStepClasses*. Thus, J is bounded by the total number of step classes in $\mathcal{M}_0 \cup \mathcal{M}_1 \cup \ldots$. (Note that each step class can serve at most once as splitter for X.) Since $\mathcal{M}_0, \mathcal{M}_1, \ldots$ can be viewed as a sequence of partitions of the set $Z = \bigcup_{s \in S} \mathit{Steps}(s)$ where $\mathcal{M}_j$ is finer than $\mathcal{M}_{j-1}$, $j = 1, 2, \ldots$ we get $J \leq |\mathcal{M}_0 \cup \mathcal{M}_1 \cup \ldots| \leq 2(m-1)$. ■

Lemma 4.6 *Let AllNewBlocks be the set of all blocks C that are once inserted into NewBlocks (i.e. the blocks for which there is procedure call $\mathit{Split}(\mathcal{M}, C)$). Then, we have:*

$$|\mathit{AllNewBlocks}| \leq 2(n-1) \quad \textit{and} \quad \sum_{C \in \mathit{AllNewBlocks}} |C| \leq n \log n.$$

Proof: Let $X_0 = X_{init}$ and X_j the state partition X at the end of the j-th iteration. Then, *AllNewBlocks* is a subset of $X_0 \cup X_1 \cup \ldots$. Hence, $|\mathit{AllNewBlocks}| \leq |X_0 \cup X_1 \cup \ldots| \leq 2(n-1)$. Here, we use the general observation that we made in the beginning of the proof of Lemma 4.5.

For each $B \in \mathit{AllNewBlocks}$ and state $s \in B$, let $I_B(s)$ be the number of blocks $C \in \mathit{AllNewBlocks}$ such that $s \in C$ and C is a proper subset of B. We show by induction on B that $I_B(s) \leq \log |B|$. If $|B| = 1$ then $I_B(s) = 0 = \log |B|$. Now we assume that $|B| \geq 2$. If s is not contained in a proper subblock $C \in \mathit{AllNewBlocks}$ of B we get $I_B(s) = 0$. Otherwise, there is a proper subblock C of B such that $s \in C$ and such that C is maximal under all subblocks $C' \in \mathit{AllNewBlocks}$ with $s \in C' \subseteq B$. (That is, C was generated by the operation $\mathit{Refine}(B, \alpha, M)$ which splits B into two (nonempty) subblocks. In particular, C is the smaller of the subblocks $B_{(\alpha,M)}$ and $B \setminus B_{(\alpha,M)}$.) Then, $|C| \leq |B|/2$. By induction hypothesis we get $I_C(s) \leq \log |C| \leq \log |B| - 1$. Thus, $I_B(s) = 1 + I_C(s) \leq \log |B|$. We conclude that any state s occurs in at most $\log n$ blocks $C \in \mathit{AllNewBlocks}$. Hence,

$$\sum_{C \in \mathit{AllNewBlocks}} |C| = \sum_{s \in S} |\{C \in \mathit{AllNewBlocks} : s \in C\}| \leq \sum_{s \in S} \log n = n \log n.$$

This yields the second claim. ■

We now show that the two-phased partitioning algorithm sketched in Figure 9 can be implemented in time $\mathcal{O}(mn(\log m + \log n))$ and space $\mathcal{O}(mn)$. This yields the following theorem.

Theorem 3 *The bisimulation equivalence classes of a PLTS (with n states and m transitions) can be decided in time $\mathcal{O}(mn(\log m + \log n))$ and space $\mathcal{O}(mn)$.*

The remainder of this section is concerned with the proof of Theorem 3.

Some implementation details: To obtain the desired time and space complexity we need some assumptions about the data structures. Instead of an explicit representation

of the transition relation $\rightarrow$ of the given PLTS, we suggest a representation of the sets $Steps_\alpha(s)$ as follows. Let $s_1, \ldots, s_n$ be a fixed enumeration of the states and $\alpha_1, \ldots, \alpha_k$ an enumeration of the actions. Then, we use a two-dimensional array $\texttt{Steps}[j,i]$, $j = 1, \ldots, n$, $i = 1, \ldots, k$, where, for state $s = s_j$ and action $\alpha = \alpha_i$, $\texttt{Steps}[j,i]$ is a pointer to the list of (pointers to the) distributions $\mu \in Steps_\alpha(s)$. For any distribution μ, we suggest a representation by a record with two compoents `Prob` and `Pre`. The first component `Prob` is a real array $\langle \texttt{Prob}[1], \ldots, \texttt{Prob}[n] \rangle$ that stores the probabilities $\texttt{Prob}[j] = \mu(s_j)$ for the states. The second component `Pre` is an array $\langle \texttt{Pre}[1], \ldots, \texttt{Pre}[k] \rangle$ where the i-th component $\texttt{Pre}[i]$ is a bit vector representation (i.e. a boolean array) for the set

$$Pre(\alpha_i, \mu) = \{s \in S : s \stackrel{\alpha_i}{\rightarrow} \mu\}.$$

(I.e. for the second component `Pre` of the record for a distribution μ we can use a two-dimensional boolean array $\texttt{Pre}[i,j]$, $i = 1, \ldots, k$, $j = 1, \ldots, n$, where $\texttt{Pre}[i,j] = 1$ if $s_j \stackrel{\alpha_i}{\rightarrow} \mu$.) The state partition X is represented by lists of pointers to the blocks $C \in X$ where the blocks itself are connected lists of pointers to the states in that block. The step partition $\mathcal{M}$ is given by a list of pointers to the step classes $\langle \alpha, M \rangle$ in $\mathcal{M}$ extended by a third component which is (a bit vector representation of) the set

$$Pre(\alpha, M) \;=\; \{s \in S : s \stackrel{\alpha}{\rightarrow} M\}.$$

Again, for the set M we suggest a list of pointers to the distributions $\mu \in M$. We suggest similar representations for the sets *NewBlocks* and *NewStepClasses* as for X and $\mathcal{M}$ respectively (possibly organized as FIFO-queues rather than lists). The auxiliary sets X_{old} and $\mathcal{M}_{old}$ that we used in the rough formulation of our algorithm in Figure 9 are not needed when we organize X and $\mathcal{M}$ by lists. For instance, for the operation $Split(\mathcal{M}, C)$, we run through the list for $\mathcal{M}$ and replace any step class $\langle \alpha, M \rangle$ in $\mathcal{M}$ by the step classes $\langle \alpha, M' \rangle \in Split(\langle \alpha, M \rangle, C)$.

We now explain some more details of a possible implementation and show how the use of these data structures leads to the desired (worst case) time complexity for the initialization step and the two phases in the algorithm of Figure 9.

Initialization: In Figure 10, an algorithm to compute the initial state partition $X_{init} = S/\approx_{Act}$ (where $s \approx_{Act} s'$ iff $act(s) = act(s')$) is sketched. (Moreover, the algorithm returns the initial step partition $\mathcal{M}_{init}$ and the set $NewBlocks_{init}$ which consists of all elements $C \in X_{init}$ but one the largest.) We deal with a fixed enumeration $\alpha_1, \ldots, \alpha_k$ of Act and construct a binary tree T by successively inserting nodes and edges. Each node v is labelled by

- its depth $v.depth$ in T,
- a subset $v.actions$ of Act,
- the names $v.left$ and $v.right$ of the left and right son of v in T.

In the case where v does not have a left (right) son $v.left$ ($v.right$) is undefined ($\perp$). Each node v of depth $k = |Act|$ is a leaf and is additionally labelled by

- a subset $v.states$ of S,
- a natural number $v.counter$ that counts the number of elements in $v.states$.

For $i = 1, \ldots, k$ put $M_{\alpha_i} := \emptyset$ and $P_i := \emptyset$;
$create(v_0, 0)$; $max\_blocksize := 0$;
For all $s \in S$ do
(1) $v := v_0$; $i := 0$;
(2) While $i < k$ do
- If $Steps_{\alpha_{i+1}}(s) \neq \emptyset$ then
  - $M_{\alpha_{i+1}} := M_{\alpha_{i+1}} \cup Steps_{\alpha_{i+1}}(s)$; $P_{i+1} := P_{i+1} \cup \{s\}$;
  - If $v.left = \perp$ then
    $create(w, i+1)$; $v.left := w$; $w.actions = v.actions \cup \{\alpha_{i+1}\}$;
  - $v := v.left$;
- If $Steps_{\alpha_{i+1}}(s) = \emptyset$ then
  - If $v.right = \perp$ then
    $create(w, i+1)$; $v.right := w$; $w.actions := v.actions$;
  - $v := v.right$;
- $i := i + 1$;

(3) $v.states := v.states \cup \{s\}$; $v.counter := v.counter + 1$;
(4) If $v.counter \geq max\_blocksize$ then
$Maxblock := v$; $max\_blocksize := v.counter$;
Return $X_{init} := \{v.states : v.depth = k\}$ and
$\mathcal{M}_{init} := \{\langle \alpha_i, M_{\alpha_i} \rangle : i = 1, \ldots, k\}$ where $Pre(\alpha_i, M_{\alpha_i}) = P_i$;
$NewBlocks_{init} := \{v.states : v.depth = k, v \neq Maxblock\}$.

Figure 10: Initialization of the two-phased partitioning algorithm

For each state s, we traverse the tree starting in the root. If we have reached a node of depth $i < k$ then we go to the left son if $Steps_{\alpha_{i+1}}(s) \neq \emptyset$, otherwise we go the right son. If we have reached a leaf (a node of depth k) then we insert s into the set $v.states$. We use an operation $create(v, i)$ that generates a node v with $v.depth = i$ and $v.actions = \emptyset$ and where $v.left$, $v.right$ are undefined, i.e. $v.left = \perp$ and $v.right = \perp$. If $i = k$ then $v.states = \emptyset$ and $v.counter = 0$. Clearly, the time complexity of this method is bounded by $\mathcal{O}(mn)$ as for any state s we traverse a tree of height $k = |Act| \leq m$. (Note that our data structures allow to test whether $Steps_\alpha(s) = \emptyset$ in constant time.) This yields the following lemma.

Lemma 4.7 *The initialization of the two-phased partitioning algorithm in Figure 9 can be performed in time $\mathcal{O}(mn)$.*

Phase 1: We now describe the splitting operator for the step classes $\langle \alpha, M \rangle \in \mathcal{M}$ according to a block C. For this, we propose the following method that computes the quotient space $M/\equiv_C$ (where $\mu \equiv_C \mu'$ iff $\mu(C) = \mu'(C)$). Similarly to the method for reactive PLTSs, we construct an ordered balanced tree $T_{(C,\alpha,M)}$ for the values $\mu(C)$, $\mu \in M$. That is, $T_{(C,\alpha,M)}$ is

an ordered balanced tree where each node v is labelled by a record $\langle v.key, v.states, v.distr\rangle$ where $v.key$ stands for one of the values $\mu(C)$, $\mu \in M$. The other components are suitable representations for the sets

$$v.distr = \{\mu \in M : \mu(C) = v.key\}.$$

and $v.states = \{s \in Pre(\alpha, M) : Steps_\alpha(s) \cap v.distr \neq \emptyset\}$. Let $M = \{\mu_1, \ldots, \mu_k\}$. We start with the tree $T_{(C,\alpha,M)}$ consisting of its root v_0 where $v_0.states = Pre(\alpha, \mu_1)$ and $v_0.distr = \{\mu_1\}$. Then, for $i = 2, \ldots, k$, we insert the value $\mu_i(C)$ into $T_{(C,\alpha,M)}$, possibly creating new nodes and edges and performing the necessary rebalance steps. (Moreover, we insert $Pre(\alpha, \mu_i)$ into $v.states$ and μ_i into $v.distr$ if v is the node with $v.key = \mu_i(C)$.) Then, for the final tree we have $Split(\langle\alpha, M\rangle, C) = \{\langle\alpha, M_v\rangle : v \text{ is a node in } T_{(C,\alpha,M)}\}$ where $M_v = v.distr$. The associated set $Pre(\alpha, M_v)$ of all states s where $s \xrightarrow{\alpha} M_v$ is given by the component $v.states$.

We now discuss the time complexity of the proposed technique. For this, we analyze the total cost that we get when we range over all iterations.

Lemma 4.8 *Ranging over all iterations, the computations the values $\mu(C)$ for the splitting operator $Split(\mathcal{M}, C)$ in phase 1 of the two-phased partitioning algorithm in Figure 9 causes the cost $\mathcal{O}(mn \log n)$.*

Proof: For fixed block C, we have to calculate $\mu(C) = \sum_{s \in C} \mu(s)$ for any of the m distribution $\mu \in \bigcup_\alpha \bigcup_{s \in S} Steps_\alpha(s)$. This takes $\mathcal{O}(m|C|)$ time when we use the bit vector representation for the block C. Lemma 4.6 yields the time complexity $\mathcal{O}(mn \log n)$ when we sum up over all $C \in AllNewBlocks$. ■

We now consider the time complexity caused by the construction of the trees $T_{(C,\alpha,M)}$ where we assume that the values $\mu(C)$ are already computed. Recall that *AllNewBlocks* denotes the set of all blocks that are once inserted into *NewBlocks*, i.e. the blocks for which we call the procedure $Split(\mathcal{M}, C)$.

Lemma 4.9 *Ranging over all iterations (i.e. over all $C \in AllNewBlocks$), the construction of the trees $T_{(C,\alpha,M)}$, $\langle\alpha, M\rangle \in \mathcal{M}$, in the splitting operation $Split(\mathcal{M}, C)$ in phase 1 of the two-phased partitioning algorithm in Figure 9 causes the total cost $\mathcal{O}(mn \log m)$. (Here, we neglect the cost for the computation of the values $\mu(C)$, $\mu \in M$, and just consider the cost for the construction of $T_{(C,\alpha,M)}$ by successively inserting the values $\mu(C)$ for $\mu \in M$.)*

Proof: For fixed $C \in AllNewBlocks$, let $Exec(C)$ be the set of all pairs $\langle\alpha, M\rangle$ that we split according to C (i.e. for which we construct the tree $T_{(C,\alpha,M)}$). Recall that

$$Split(\langle\alpha, M\rangle, C) = \{\langle\alpha, M'\rangle : M' \in M/\equiv_C\}$$

is the set of all pairs in which the step class $\langle\alpha, M\rangle$ is splitted when the procedure $Split(\mathcal{M}, C)$ is called. For fixed C, and step class $\langle\alpha, M\rangle$ the construction of the tree $T_{(C,\alpha,M)}$ takes $\mathcal{O}(K_{(C,\alpha,M)})$ time where

$$K_{(C,\alpha,M)} \;=\; |M| \log\Big(|T_{(C,\alpha,M)}| + 1\Big)$$

and $|T_{(C,\alpha,M)}| = |\mathit{Split}(\langle \alpha, M\rangle, C)|$ denotes the number of nodes in $T_{(C,\alpha,M)}$. Ranging over all iterations, we get the total time complexity $\mathcal{O}(K)$ where

$$\begin{aligned} K &= \sum_{C \in \mathit{AllNewBlocks}} \sum_{\langle \alpha, M\rangle \in \mathit{Exec}(C)} K_{(C,\alpha,M)} \\ &= \sum_{C \in \mathit{AllNewBlocks}} \sum_{\langle \alpha, M\rangle \in \mathit{Exec}(C)} |M| \log\Big(|T_{(C,\alpha,M)}| + 1\Big). \end{aligned}$$

We now show that $K \leq mn \log m$. Let $C_1, C_2, \ldots, C_J$ be the order of the blocks $C \in \mathit{AllNewBlocks}$ in which the procedure $\mathit{Split}(\mathcal{M}, C)$ is called during the whole execution of the two-phased partitioning algorithm. Then, for $j < J$ we have

$$\mathit{Exec}(C_{j+1}) = \bigcup_{\langle \alpha, M\rangle \in \mathit{Exec}(C_j)} \mathit{Split}(\langle \alpha, M\rangle, C_j).$$

For $\langle \alpha, M\rangle \in \mathit{Exec}(C_j)$, we define $K'_{(j,\alpha,M)}$ as follows. Let $K'_{(J,\alpha,M)} = K_{(J,\alpha,M)}$. If $j < J$ then

$$K'_{(j,\alpha,M)} = |M| \log\Big(|T_{(C_j,\alpha,M)}| + 1\Big) + \sum_{\langle \alpha, M'\rangle \in \mathit{Split}(\langle \alpha, M\rangle, C_j)} K'_{(j+1,\alpha,M')}.$$

Then, $\mathcal{O}(K'_{(j,\alpha,M)})$ is an upper bound for the asymptotic cost for the construction of the trees $T_{(C_l,\alpha,M')}$ where $M' \subseteq M$ and $\langle \alpha, M'\rangle \in \mathit{Exec}(C_l)$, $l = j, j+1, \ldots, J$. By induction on j we show that

$$K'_{(J-j,\alpha,M)} \leq (j+1)|M| \log(|M| + 1)$$

for all $\langle \alpha, M\rangle \in \mathit{Exec}(J - j)$.

- In the basis of induction ($j = 0$) we have $|T_{(C_J,\alpha,M)}| \leq |M|$. Hence,

$$K'_{(J,\alpha,M)} \leq |M| \log(|M| + 1).$$

- Induction step: Let $1 \leq j \leq J - 1$, $\langle \alpha, M\rangle \in \mathit{Exec}(J - j)$ and $\mathit{Split}(\langle \alpha, M\rangle, C_{J-j}) = \{\langle \alpha, M_1\rangle, \ldots, \langle \alpha, M_r\rangle\}$. By induction hypothesis,

$$K'_{(J-j+1,\alpha,M_i)} \leq j \cdot |M_i| \log(|M_i| + 1), \quad i = 1, \ldots, r.$$

Since $|M_1| + \ldots + |M_r| \leq |M|$ and $|T_{(C_{J-j},\alpha,M)}| \leq |M|$ we get

$$\begin{aligned} K'_{(J-j,\alpha,M)} &\leq |M| \log\Big(|T_{(C_{J-j},\alpha,M)}| + 1\Big) + j \cdot \sum_{i=1}^{r} |M_i| \log(|M_i| + 1) \\ &\leq |M| \log(|M| + 1) + j \cdot |M| \log(|M| + 1) = (j+1)|M| \log(|M| + 1). \end{aligned}$$

Since $\mathit{Exec}(C_1) = \{\langle \alpha, M_\alpha\rangle : \alpha \in \mathit{Act}\}$ we get

$$\sum_{\langle \alpha, M\rangle \in \mathit{Exec}(C_1)} |M| = \sum_{\alpha \in \mathit{Act}} |M_\alpha| = m.$$

Hence (with the above claim for $j = J - 1$),

$$K = \sum_{\langle \alpha, M\rangle \in \mathit{Exec}(C_1)} K'_{(C_1,\alpha,M)} \leq \sum_{\alpha \in M_\alpha} J \cdot |M_\alpha| \log |M_\alpha| \leq 2nm \log m$$

where we use the fact that the total number J of blocks in *AllNewBlocks* is bounded by $2(n-1)$ (Lemma 4.6). ■

Combining Lemma 4.8 and 4.9 yields:

Lemma 4.10 *Ranging over all iterations, the operations Split$(\mathcal{M}, C)$ in phase 1 of the two-phased partitioning algorithm (Figure 9) can be implemented in time $\mathcal{O}(mn(\log m + \log n))$.*

Phase 2: We now analyse the cost caused by the operations *Refine*(X, α, M). Given a step class $\langle \alpha, M \rangle$ in *NewStepClasses*, to perform the refinement step *Refine*(X, α, M) we run through the list for X and replace any block $B \in X$ by the blocks obtained by *Refine*(B, α, M). For the refinement of a block $B \in X$ according to a step class $\langle \alpha, M \rangle \in \mathcal{M}$ (the operator *Refine*(B, α, M)), we just have to calculate the sets $B_{(\alpha,M)} = B \cap Pre(\alpha, M)$ and $B \setminus B_{(\alpha,M)} = B \setminus Pre(\alpha, M)$.

Lemma 4.11 *The operator Refine(X, α, M) takes $\mathcal{O}(n)$ time.*

Proof: To perform *Refine*(B, α, M), we just have to run through the list for B and check for any state $s \in B$ whether s belongs to $Pre(\alpha, M)$. If $s \in Pre(\alpha, M)$ then we insert s into the list for $B_{(\alpha,M)}$; otherwise we add s to the list for $B \setminus B_{(\alpha,M)}$. Clearly, with this technique the operation *Refine*(B, α, M) takes $\mathcal{O}(|B|)$ time. (Recall that for any step class $\langle \alpha, M \rangle$ in *NewStepClasses*, we have an additional component that yields the bit vector representation for the set $Pre(\alpha, M) = \{s \in S : s \xrightarrow{\alpha} M\}$. Thus, given a state s, we can test in constant time whether $s \in Pre(\alpha, M)$.) Summing up over all $B \in X$, we get the time complexity $\mathcal{O}(n)$ for *Refine*(X, α, M). ■

Lemma 4.12 *Ranging over all iterations, the total time complexity for phase 2 of the two-phased partitioning algorithm in Figure 9 is $\mathcal{O}(mn)$.*

Proof: The total number of step classes $\langle \alpha, M \rangle$ that will be once in *NewStepClasses* is bounded by the number $|\mathcal{M}_0 \cup \mathcal{M}_1 \cup \ldots| \leq 2(m-1)$; see Lemma 4.5. Lemma 4.11 yields the time complexity $\mathcal{O}(mn)$ for all refinement operations in phase 2 together. ■

Proof of Theorem 3: In summary, the time complexity of the two-phased partitioning algorithm in Figure 9 is $\mathcal{O}(mn(\log m + \log n))$ (Lemma 4.7, 4.10 and 4.12). Clearly, with the suggested data structures we need $\mathcal{O}(mn)$ space for the representation of the given PLTS and the partitions X, $\mathcal{M}$ (and the subsets *NewBlocks* and *NewStepClasses*). The number of nodes of the trees $T_{(C,\alpha,M)}$ is bounded by $|M|$; and hence at most m. As we need the trees $T_{(C,\alpha,M)}$ only temporary and as each node requires $\mathcal{O}(n)$ space, our tree-based implementation of phase 1 does not exceed the space complexity $\mathcal{O}(mn)$. For the refinement operation *Refine*(X, α, M) in phase 2, we do not need additional space. This yields the space complexity stated in Theorem 3. ■

Example 4.13 We consider the PLTS of Figure 11 and compute the bisimulation equivalence classes with the two-phased partitioning algorithm. In the initialization (the algorithm in Figure 10), we use the ordering $\alpha_1 = \alpha$, $\alpha_2 = \beta$, $\alpha_3 = \gamma$ of *Act* and construct a tree as shown in Figure 12. We obtain $X_{init} = \{C_1, C_2, C_3, C_4\}$, $NewBlocks_{init} = \{C_1, C_2, C_3\}$ and

$$\mathcal{M}_{init} \;=\; NewStepClasses \;=\; \{\langle \alpha, M_\alpha \rangle, \langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\}$$

where $M_\alpha = \{\mu_1, \mu_2, \mu_4, \mu^1_{u_1}, \mu^1_{u_2}, \mu^1_{t_3}\}$, $M_\beta = \{\mu^1_{v_1}, \mu^1_{v_2}, \mu^1_{v_3}, \mu^1_{v_4}\}$ and $M_\gamma = \{\mu^1_v\}$. Moreover, we have $Pre(\alpha, M_\alpha) = \{s_1, s_2, s_3, s_4\} = C_1$, $Pre(\beta, M_\beta) = \{t_1, t_2, t_3, t_4\} = C_2$ and

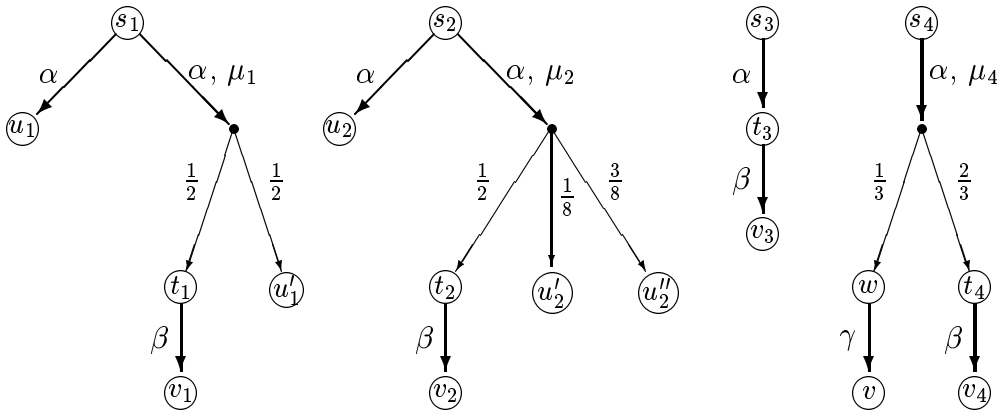
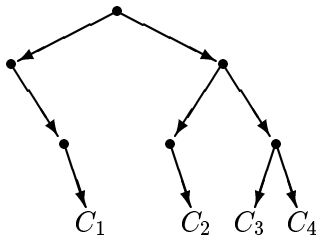


Figure 11:



$$\begin{aligned} C_1 &= \{s_1, s_2, s_3, s_4\} \\ C_2 &= \{t_1, t_2, t_3, t_4\} \\ C_3 &= \{w\} \\ C_4 &= \{u_1, u_1', u_2, u_2', u_2'', v, v_1, v_2, v_3, v_4\} \end{aligned}$$

Figure 12: The tree constructed in the initialization step

$\mathit{Pre}(\gamma, M_\gamma) = \{w\} = C_3$. In the first iteration, phase 1 takes one of the blocks in *NewBlocks*, say block $C = C_1 = \{s_1, s_2, s_3, s_4\}$, and calls the procedure $\mathit{Split}(\mathcal{M}, C_1)$ which tries to split any step class in $\mathcal{M} = \mathcal{M}_{init}$ according to C_1. For any of the step classes $\langle \alpha, M_\alpha \rangle$, $\langle \beta, M_\beta \rangle$ and $\langle \gamma, M_\gamma \rangle$, the constructed ordered balanced tree just consists of a single node and hence yields no proper splitting. For example, for the step class $\langle \alpha, M_\alpha \rangle$ we construct an ordered balanced tree $T_{(C_1, \alpha, M_\alpha)}$ for the values

$$\mu^1_{u_1}(C_1) = \mu^1_{u_2}(C_1) = \mu^1_{t_3}(C_1) = \mu_4(C_1) = \mu_1(C_1) = \mu_2(C_1) = 0$$

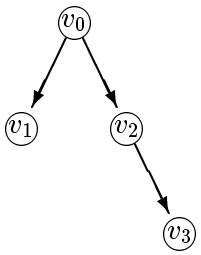
which yields a tree consisting of its root. (For the root v_0 of $T_{(C_1, \alpha, M_\alpha)}$ we have $v_0.\mathit{distr} = M_\alpha$ and $v_0.\mathit{states} = \mathit{Pre}(\alpha, M_\alpha)$. Hence, we get $\mathit{Split}(\langle \alpha, M_\alpha \rangle, C_1) = \{\langle \alpha, M_\alpha \rangle\}$.) Then, in phase 2, we take a step class of *NewStepClasses*, say $\langle \alpha, M_\alpha \rangle$, and apply the operator $\mathit{Refine}(X_{init}, \alpha, M_\alpha)$ which devides any of the blocks C_i into the subblocks $C_i \cap \mathit{Pre}(\alpha, M_\alpha)$ and $C_i \setminus \mathit{Pre}(\alpha, M_\alpha)$ (where we neglect the empty blocks). This operation is redundant as it yields $X_{init} = \mathit{Refine}(X_{init}, \alpha, M_\alpha)$. Thus, after the first iteration we have

$$\begin{array}{lcllcl} X &=& \{C_1, C_2, C_3, C_4\}, & \mathit{NewBlocks} &=& \{C_2, C_3\}, \\ \mathcal{M} &=& \{\langle \alpha, M_\alpha \rangle, \langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\}, & \mathit{NewStepClasses} &=& \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\}. \end{array}$$

Let us assume that we choose the block $C_2 = \{t_1, t_2, t_3, t_4\} \in \mathit{NewBlocks}$ in phase 1 of the second iteration. Then, we calculate the step partition $\mathit{Split}(\mathcal{M}, C_2)$. For instance, for the step class $\langle \alpha, M_\alpha \rangle$, we construct an ordered balanced tree $T_{(C_2, \alpha, M_\alpha)}$ for the values

$$\mu^1_{u_1}(C_2) = \mu^1_{u_2}(C_2) = 0, \quad \mu^1_{t_3}(C_2) = 1, \quad \mu_4(C_2) = \tfrac{2}{3}, \quad \mu_1(C_2) = \mu_2(C_2) = \tfrac{1}{2}$$

which might be the tree as shown in Figure 13. Hence, $\mathit{Split}(\langle \alpha, M_\alpha \rangle, C_2) = \{\langle \alpha, M_{\alpha,i} \rangle :$



$$
\begin{array}{lcllcl}
v_0.key & = & 1/2 & v_0.states & = & \{s_1, s_2\} \\
 & & & v_0.distr & = & \{\mu_1, \mu_2\} \\
v_1.key & = & 0 & v_1.states & = & \{s_1, s_2\} \\
 & & & v_1.distr & = & \{\mu^1_{u_1}, \mu^1_{u_2}\} \\
v_2.key & = & 2/3 & v_2.states & = & \{s_4\} \\
 & & & v_2.distr & = & \{\mu_4\} \\
v_3.key & = & 1 & v_3.states & = & \{s_3\} \\
 & & & v_3.distr & = & \{\mu^1_{t_3}\}
\end{array}
$$

Figure 13: The ordered balanced tree $T_{(C_2,\alpha,M_\alpha)}$

$i = 1, 2, 3, 4\}$ where

$$
\begin{array}{lcllcl}
M_{\alpha,1} & = & \{\mu_1, \mu_2\}, & \mathit{Pre}(\alpha, M_{\alpha,1}) & = & \{s_1, s_2\}, \\
M_{\alpha,2} & = & \{\mu^1_{u_1}, \mu^1_{u_2}\}, & \mathit{Pre}(\alpha, M_{\alpha,2}) & = & \{s_1, s_2\}, \\
M_{\alpha,3} & = & \{\mu^1_{t_3}\}, & \mathit{Pre}(\alpha, M_{\alpha,3}) & = & \{s_3\}, \\
M_{\alpha,4} & = & \{\mu_4\}, & \mathit{Pre}(\alpha, M_{\alpha,1}) & = & \{s_4\}.
\end{array}
$$

The new step classes $\langle \alpha, M_{\alpha,i} \rangle$, $i = 1, 2, 3, 4$, are inserted into *NewStepClasses*. The operations $\mathit{Split}(\langle \beta, M_\beta \rangle, C_2)$ and $\mathit{Split}(\langle \gamma, M_\gamma \rangle, C_2)$ do not yield new step classes. In phase 2 of the second iteration, we may take the step class $\langle \beta, M_\beta \rangle$ which yields $X = \mathit{Refine}(X, \beta, M_\beta)$ for the current partition $X = \{C_1, C_2, C_3, C_4\}$. In summary, the second iteration yields

$$
\begin{array}{l}
X = \{C_1, C_2, C_3, C_4\}, \quad \mathit{NewBlocks} = \{C_3\}, \\
\mathcal{M} = \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1, 2, 3, 4\}, \\
\mathit{NewStepClasses} = \{\langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1, 2, 3, 4\}.
\end{array}
$$

In the third iteration, we (try to) split $\mathcal{M}$ according to the block $C_3 \in \mathit{NewBlocks}$ (which yields $\mathit{Split}(\mathcal{M}, C_3) = \mathcal{M}$ and $\mathit{NewBlocks} = \emptyset$). When we choose the step class $\langle \gamma, M_\gamma \rangle$ in phase 2, then the total effect of the third iteration is

$$
\begin{array}{l}
X = \{C_1, C_2, C_3, C_4\}, \quad \mathit{NewBlocks} = \emptyset, \\
\mathcal{M} = \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1, 2, 3, 4\}, \\
\mathit{NewStepClasses} = \{\langle \alpha, M_{\alpha,i} \rangle : i = 1, 2, 3, 4\}.
\end{array}
$$

In the fourth iteration, we skip phase 1 (as $\mathit{NewBlocks} = \emptyset$). Let $\langle \alpha, M_{\alpha,1} \rangle$ be the step class that is chosen in phase 2 of the fourth iteration. The operator $\mathit{Refine}(X, \alpha, M_{\alpha,1})$ devides the block $C_1 = \{s_1, s_2, s_3, s_4\}$ into the two proper subblocks

$$C_{1,1} = C_1 \cap \mathit{Pre}(\alpha, M_{\alpha,1}) = \{s_1, s_2\}, \quad C_{1,2} = C_1 \setminus \mathit{Pre}(\alpha, M_{\alpha,1}) = \{s_3, s_4\}$$

and inserts one of them into *NewBlocks*. Thus, after the fourth iteration we might have

$$
\begin{array}{l}
X = \{C_{1,1}, C_{1,2}, C_3, C_4\}, \quad \mathit{NewBlocks} = \{C_{1,1}\}, \\
\mathcal{M} = \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1, 2, 3, 4\}, \\
\mathit{NewStepClasses} = \{\langle \alpha, M_{\alpha,i} \rangle : i = 2, 3, 4\}.
\end{array}
$$

Splitting $\mathcal{M}$ according to $C_{1,1}$ (in phase 1 of the fifth iteration) has no effect. Also the choice of the step class $\langle \alpha, M_{\alpha,2} \rangle$ as splitter for the refinement operation in phase 2 of the

fifth iteration does not change the state partition X. Thus, after the fifth iteration we might have

$$\begin{array}{lcl} X & = & \{C_{1,1}, C_{1,2}, C_3, C_4\}, \quad \mathit{NewBlocks} \;=\; \emptyset, \\ \mathcal{M} & = & \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1,2,3,4\}, \\ \mathit{NewStepClasses} & = & \{\langle \alpha, M_{\alpha,i} \rangle : i = 3,4\}. \end{array}$$

As *NewBlocks* is empty, phase 1 of the sixth iteration is skipped. The refinement operation $\mathit{Refine}(X, \alpha, M_{\alpha,3})$ in phase 2 of the sixth iteration devides the block $C_{1,2} = \{s_3, s_4\}$ into the subblocks

$$C_{1,2,1} = C_{1,2} \cap \mathit{Pre}(\alpha, M_{\alpha,3}) = \{s_3\}, \quad C_{1,2,2} = C_{1,2} \setminus \mathit{Pre}(\alpha, M_{\alpha,3}) = \{s_4\}.$$

Thus, we start the seventh iteration e.g. with

$$\begin{array}{lcl} X & = & \{C_{1,1}, C_{1,2,1}, C_{1,2,2}, C_3, C_4\}, \quad \mathit{NewBlocks} \;=\; \{C_{1,2,1}\}, \\ \mathcal{M} & = & \{\langle \beta, M_\beta \rangle, \langle \gamma, M_\gamma \rangle\} \cup \{\langle \alpha, M_{\alpha,i} \rangle : i = 1,2,3,4\}, \\ \mathit{NewStepClasses} & = & \{\langle \alpha, M_{\alpha,i} \rangle : i = 4\}. \end{array}$$

As $\mathit{Split}(\mathcal{M}, C_{1,2,1}) = \mathcal{M}$ and $\mathit{Refine}(X, \alpha, M_{\alpha,4}) = X$, the seventh iteration yields

$$\mathit{NewBlocks} = \mathit{NewStepClasses} = \emptyset.$$

The algorithm terminates and returns the state partition

$$X = \{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \{t_1, t_2, t_3, t_4\}, C_4\}$$

(where, as before, C_4 collects the terminal states). ■

5 Computing the simulation preorder

In this section, we present an algorithm that computes the simulation preorder of a given PLTS. As before, we fix a PLTS $(S, \mathit{Act}, \rightarrow)$ and use the notations as explained in Notation 4.1.

The general schema: The key idea of our algorithm is as in the non-probabilistic case [38] (see Figure 14): we start with the trivial preorder $R = S \times S$ and then successively remove those pairs (s, s') from R where $s \not\sqsubseteq_R s'$. Intuitively, the condition $s \not\sqsubseteq_R s'$ asserts that s has a transition that cannot be "simulated" by a transition of s' where "simulation" is understood with respect to the current relation R. The formal definition of the relation $\sqsubseteq_R$ is as follows. If $R \subseteq S \times S'$ and s, $s' \in S$, we define $s \sqsubseteq_R s'$ iff for each $\alpha \in \mathit{Act}$ and $\mu \in \mathit{Steps}_\alpha(s)$ there exists $\mu' \in \mathit{Steps}_\alpha(s')$ with $\mu \sqsubseteq_R \mu'$.

Our algorithm relies on an explicit test for the condition $s \sqsubseteq_R s'$ with a network-based technique for checking whether $\mu \sqsubseteq_R \mu'$. This stands in contrast to the method proposed by Henzinger, Henzinger & Kopke [38] for ordinary LTSs. Although [38] also uses the schema of Figure 14, the algorithm by [38] does not work with an explicit test for $s \sqsubseteq_R s'$. Instead, it successively removes those pairs (s, s') of R where s is an α-predecessor of some state t (in the sense that there is a transition $s \stackrel{\alpha}{\rightarrow} t$) and s' does not have an α-successor

$R := S \times S$;
While there exists $(s, s') \in R$ with $s \not\sqsubseteq_R s'$ do
$R := R \setminus \{(s, s')\}$;
Return R.

Figure 14: Schema for computing the simulation preorder

in $\{t' : (t, t') \in R\}$. However, it seems to be hard to modify the ideas of [38] for the probabilistic case because the induced predecessor/successor relations on the states of a PLTS (e.g. s is an α-predecessor of t iff $\mu(t) > 0$ for some $\mu \in \mathit{Steps}_\alpha(s)$) does not give enough information. Even the probabilities for the α-successors/predecessors of the states do not contain the necessary information for computing the simulation preorder since there might be non-similar states that cannot be distinguished with these predecessor/successor relations (cf. Remark 3.9).

The test whether $\mu \sqsubseteq_R \mu'$: We show that the question whether $\mu \sqsubseteq_R \mu'$ can be reduced to a maximum flow problem in a suitable chosen network. (See Section 2 for our notations that we use for networks.) Let R a subset of $S \times S$ and μ, $\mu' \in \mathit{Distr}(S)$. Let $\overline{S} = \{\overline{t} : t \in S\}$ where $\overline{t}$ are pairwise distinct "new" states (i.e. $\overline{t} \notin S$). We choose new elements $\bot$ and $\top$ not contained in $S \cup \overline{S}$, $\bot \neq \top$. We associate with (μ, μ') the following network $\mathcal{N}(\mu, \mu', R)$. The nodes are the elements of $S \cup \overline{S}$ and $\bot$ (the source) and $\top$ (the

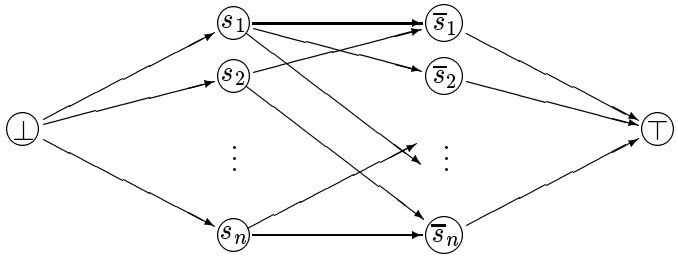


Figure 15: The graph (N, E)

sink), i.e. $N = \{\bot, \top\} \cup S \cup \overline{S}$. The edges are

$$E = \{(s, \overline{t}) : (s, t) \in R\} \cup \{ (\bot, s) : s \in S \} \cup \{ (\overline{t}, \top) : t \in S \}.$$

The graph (N, E) is shown in Figure 15 where $S = \{s_1, \ldots, s_n\}$ and where the arrows between the nodes s_i and the nodes $\overline{s}_j$ describe the relation R in the sense that there is an arrow from s_i to $\overline{s}_j$ iff $(s_i, s_j) \in R$. The capacities $c(e) \in [0, 1]$ are given by: $c(\bot, s) = \mu(s)$, $c(\overline{t}, \top) = \mu'(t)$, $c(s, \overline{t}) = 1$.

Lemma 5.1 *The following are equivalent:*

(i) There exists a weight function δ for (μ, μ') w.r.t. R.

(ii) The maximum flow in $\mathcal{N}(\mu, \mu', R)$ is 1.

Proof: (i) $\Longrightarrow$ (ii): For each flow function f in $\mathcal{N}(\mu, \mu', R)$:

$$\mathcal{F}(f) \;=\; \sum_{s \in S} f(\bot, s) \;\leq\; \sum_{s \in S} c(\bot, s) \;=\; \sum_{s \in S} \mu(s) \;=\; 1.$$

Let δ be a weight function for (μ, μ') w.r.t. R. Then we define a flow function f as follows: $f(\bot, s) = \mu(s)$, $f(\overline{t}, \top) = \mu'(t)$, $f(s, \overline{t}) = \delta(s, t)$. Then, $\mathcal{F}(f) = 1$. Hence, the maximum flow of $\mathcal{N}(\mu, \mu', R)$ is 1.

(ii) $\Longrightarrow$ (i): Let f be a flow function with $\mathcal{F}(f) = 1$. Since $f(\bot, s) \leq c(\bot, s) = \mu(s)$ and since

$$\sum_{s \in S} f(\bot, s) \;=\; \mathcal{F}(f) \;=\; 1 \;=\; \sum_{s \in S} \mu(s)$$

we get $f(\bot, s) = \mu(s)$ for all $s \in S$. Similarly, we get $f(\overline{t}, \top) = \mu'(t)$ for all $t \in S$. Let $\delta(s, t) = f(s, \overline{t})$ for all $(s, t) \in R$ and $\delta(s, t) = 0$ if $(s, t) \notin R$. Then,

$$\sum_{t \in S} \delta(s, t) \;=\; \sum_{t \in S} f(s, \overline{t}) \;=\; f(\bot, s) \;=\; \mu(s)$$

and similarly, $\sum_{s \in S} \delta(s, t) = \mu'(t)$. Hence, δ is a weight function for (μ, μ') w.r.t. R. ■

Lemma 5.1 yields a method for deciding whether $\mu \sqsubseteq_R \mu'$. We construct the network $\mathcal{N}(\mu, \mu', R)$ and compute the maximum flow with well-known methods (see Figure 16). To

Input: a nonempty, finite set S, distributions μ, $\mu' \in \mathit{Distr}(S)$ and $R \subseteq S \times S$
Output: If $\mu \sqsubseteq_R \mu'$ then "Yes" else "No"
Method:
Construct the network $\mathcal{N}(\mu, \mu', R)$;
Compute the maximum flow F in $\mathcal{N}(\mu, \mu', R)$;
If $F < 1$ then return "No" else return "Yes".

Figure 16: Test whether $\mu \sqsubseteq_R \mu'$

our knowledge, the best known algorithm for computing the maximum flow in a network is those of [15] which has time and space complexity $\mathcal{O}(n^3 / \log n)$ and $\mathcal{O}(n^2)$ respectively where n is the number of nodes in the network. Hence:

Lemma 5.2 *The test whether $\mu \sqsubseteq_R \mu'$ can be done in time $\mathcal{O}(n^3 / \log n)$ and space $\mathcal{O}(n^2)$.*

Example 5.3 We consider the PLTS of Example 3.7 (Figure 2), the distributions μ, μ' and the relation $R = \{(t, t'), (u, u'), (u, t')\}$. The associated network $\mathcal{N}(\mu, \mu', R)$ has the form as shown in Figure 17. (We neglect the nodes $x \in S$ with $\mu(x) = 0$ and the nodes $\overline{y} \in \overline{S}$ with $\mu'(y) = 0$.) The flow function f with $f(\bot, t) = \mu(t) = 1/3$, $f(\bot, u) = \mu(u) = 2/3$, $f(t, \overline{t}') = 1/3$, $f(u, \overline{t}') = 1/6$, $f(u, \overline{u}') = 1/2$, $f(\overline{u}', \top) = \mu'(u') = 1/2$ and $f(\overline{t}', \top) = \mu'(t') = 1/2$ yields the maximum flow $\mathcal{F}(f) = 1$. Thus, $\mu \sqsubseteq_R \mu'$. ■

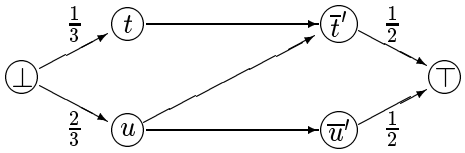


Figure 17:

Initialization:

$R := \{(s, s') \in S \times S : act(s) \subseteq act(s')\}$;

For all $(s, s') \in R$ and $s \xrightarrow{\alpha} \mu$ do $Sim_{(s,\alpha,\mu)}(s') := Steps_{\alpha}(s')$;

Iteration:

Repeat

$R_{old} := R$; $R := \emptyset$;

For all $(s, s') \in R_{old}$ do

- $sim := true$;
- For all $s \xrightarrow{\alpha} \mu$ do
  - * Repeat

    choose some $\mu' \in Sim_{(s,\alpha,\mu)}(s')$;

    If $\mu \not\sqsubseteq_R \mu'$ then remove μ' from $Sim_{(s,\alpha,\mu)}(s')$;

    until $Sim_{(s,\alpha,\mu)}(s') = \emptyset$ or $\mu \sqsubseteq_R \mu'$;
  - * If $Sim_{(s,\alpha,\mu)}(s') = \emptyset$ then $sim := false$;
- If $\neg sim$ then $R := R \cup \{(s, s')\}$;

until $R_{old} = R$;

Output: Return R.

Figure 18: Basic algorithm for computing the simulation preorder in PLTSs

Remark 5.4 Another possibility for testing whether $\mu \sqsubseteq_R \mu'$ is to consider the following linear inequality system with the variables $x_{s,t}$, $(s, t) \in R$:

$$x_{s,t} \geq 0, \quad \text{for all } (s,t) \in R, \quad \sum_{\substack{t \in S \\ (s,t) \in R}} x_{s,t} \;=\; \mu(s), \qquad \sum_{\substack{t \in S \\ (t,s) \in R}} x_{s,t} \;=\; \mu'(s), \quad \text{for all } s \in S$$

Then, $\mu \sqsubseteq_R \mu'$ iff the system above has a solution. In that case, the solution $(x_{s,t})_{(s,t)\in R}$ yields a weight function for (μ, μ') w.r.t. R. The above system has $|R| = \mathcal{O}(n^2)$ variables and $|R| + 2|S| = \mathcal{O}(n^2)$ equations. To our knowledge, there is no method for solving inequality systems of this type that beat the time complexity $\mathcal{O}(n^3/\log n)$. ■

Basic algorithm for computing the simulation preorder: In Figure 18, we formulate the basic schema of our algorithm for computing the simulation preorder in a given PLTS. The key idea is the schema of Figure 14 except that we start with the relation

$$R_{init} = \{(s, s') \in S \times S : act(s) \subseteq act(s'), s \neq s'\}$$

and keep book about the distributions μ, μ' where $\mu \not\sqsubseteq_R \mu'$ is already detected. The fact that we start with R_{init} defined as above (rather than $R_{init} = S \times S$) is motivated by the observation that for the "trivial" pairs (s, s) we always have $s \sqsubseteq s$. To avoid unnecessary tests for $\mu \sqsubseteq_R \mu'$, we use the fact that if once $\mu \not\sqsubseteq_R \mu'$ is detected then $\mu \not\sqsubseteq_R \mu'$ in all further iterations. To store the information which tests $\mu \sqsubseteq_R \mu'$ have still be done we suggest the following. For any pair $(s, s') \in R$ and transition $s \xrightarrow{\alpha} \mu$, we deal with a set

$$Sim_{(s,\alpha,\mu)}(s') \subseteq Steps_\alpha(s')$$

that contains all those distributions $\mu' \in Steps_\alpha(s')$ that are (still) possible candidates for "matching" the transition $s \xrightarrow{\alpha} \mu$. That is, a distribution $\mu' \in Steps_\alpha(s')$ belongs to $Sim_{(s,\alpha,\mu)}(s')$ iff $\mu \not\sqsubseteq_R \mu'$ is not yet verified (either because we did not yet check whether $\mu \sqsubseteq_R \mu'$ or the last test for $\mu \sqsubseteq_R \mu'$ was successful).

- Initially, $Sim_{(s,\alpha,\mu)}(s') = Steps_\alpha(s')$ for all $(s, s') \in R_{init}$ and transitions $s \xrightarrow{\alpha} \mu$.
- A distribution $\mu' \in Steps_\alpha(s')$ is removed from $Sim_{(s,\alpha,\mu)}(s')$ just in the moment where $\mu \not\sqsubseteq_R \mu'$ is detected. (Then, $\mu \not\sqsubseteq_R \mu'$ in all further iterations.)

To check whether $s \sqsubseteq_R s'$ for the current relation R, we consider all transitions $s \xrightarrow{\alpha} \mu$ and search in $Sim_{(s,\alpha,\mu)}(s')$ for a distribution μ' with $\mu \sqsubseteq_R \mu'$ (which we test by computing the maximum flow in $\mathcal{N}(\mu, \mu', R)$ as described before). We use the boolean variable *sim* which is set to *false* as soon as we have found a transition $s \xrightarrow{\alpha} \mu$ where there is no corresponding transition from s' (i.e. *sim* stands for the truth value of the condition $s \not\sqsubseteq_R s'$).

The following lemma yields the total correctness of our algorithm. Moreover, it states an upper bound for total the number of procedure calls to check whether $\mu \sqsubseteq_R \mu'$. (Here, we range over all iterations. By a *successful test*, we mean a procedure call to check whether $\mu \sqsubseteq_R \mu'$ which yields the answer "Yes". Similarly, we speak about an *unsuccessful test* for $\mu \sqsubseteq_R \mu'$ if the algorithm that checks whether $\mu \sqsubseteq_R \mu'$ gives the answer "No".)

Lemma 5.5 *The algorithm in Figure 18 halts after at most n^2 iterations and returns the simulation preorder. Moreover:*

(a) There are at most $\sum_{\alpha \in Act} m_\alpha^2$ unsuccessful tests whether $\mu \sqsubseteq_R \mu'$.
(b) There are at most mn^3 successful tests whether $\mu \sqsubseteq_R \mu'$.

Proof: Let $R_0 = R_{init}$ and R_j the relation R after the j-th iteration. Then, $R_0, R_1, R_2, \ldots$ is a decreasing sequence of subsets of $S \times S$. Thus, there is some $J \leq |S \times S| = n^2$ with $R_0 \supset R_1 \supset \ldots \supset R_{J-1} = R_J$. After the J-th iteration, the algorithm returns R_J and terminates. We now show that $R = R_J$ is the simulation preorder. The fact that $R_{J-1} = R_J (= R)$ yields that in the last iteration:

If $(s, s') \in R$ and $s \xrightarrow{\alpha} \mu$ then there is some $\mu' \in Sim_{(s,\alpha,\mu)}(s') \subseteq Steps_\alpha(s')$ with $\mu \sqsubseteq_R \mu'$.

Thus, R is a simulation which yields $R \subseteq \sqsubseteq$. Vice versa, it easy to see by induction on j that R_j is coarser than $\sqsubseteq$. For this, we use the fact that at any moment $Sim_{(s,\alpha,\mu)}(s')$ is a superset of $\{\mu' \in Steps_\alpha(s') : \mu \sqsubseteq_{R_j} \mu'\}$. (This follows from the observation that $\mu \not\sqsubseteq_{R_j} \mu'$ implies $\mu \not\sqsubseteq_{R_{j+1}} \mu'$.) This yields that the final relation $R = R_J$ coincides with the simulation preorder.

Next we prove the subclaim (a). As for all $(s, s') \in S \times S$ and actions $\alpha \in Act$, for each pair (μ, μ') of distrubutions $\mu \in Steps_\alpha(s)$ and $\mu' \in Steps_\alpha(s')$ there is at most one test whether $\mu \sqsubseteq_R \mu'$ for the current relation R that returns the answer "No". Thus, the total number of unsuccessful tests for $\mu \sqsubseteq_R \mu'$ is bounded by $\sum_{\alpha \in Act} m_\alpha^2$.

For subclaim (b), we use the following argument. For any state s and transition $s \xrightarrow{\alpha} \mu$, in each of the J iterations, for any state s' where (s, s') belongs to the current relation R there is at most one distribution $\mu' \in Sim_{(s,\alpha,\mu)}(s')$ for which we call the procedure call to check whether $\mu \sqsubseteq_R \mu'$ returns a positive answer. Thus, in any iteration, the number of successful tests is bounded by $\sum_{s \in S} m_s \cdot n = mn$. Summing up over all iterations yields mn^3 as an upper bound for the total number of successful tests for $\mu \sqsubseteq_R \mu'$. ■

Theorem 4 *The simulation preorder of a PLTS (with n states and m transitions) can be computed in time $\mathcal{O}\left((mn^6 + m^2n^3)/\log n\right)$ and space $\mathcal{O}\left(mn + n^2 + m^2\right)$.*

The complexity result stated in Theorem 4 can be obtained by a suitable implementation of our basic algorithm (Figure 18). Here, we assume appropriate data structures (that we describe below and that yield the tighter space complexity $\mathcal{O}\left(mn + n^2 + \sum_\alpha m_\alpha^2\right)$) and use the observation that $\mathcal{O}(((mn^6 + m^2n^3)/\log n)$ is an asymptotic upper bound for the amount of time that is needed for the procedure calls that test whether $\mu \sqsubseteq_R \mu'$. (The latter follows by combining the results of Lemma 5.2 and parts (a) and (b) of Lemma 5.5 which yields the tighter time bound $\mathcal{O}\left((mn^6 + \sum_\alpha m_\alpha^2 n^3)/\log n\right)$.)

In reactive PLTSs, we have $m_\alpha \leq n$ and $m \leq |Act| \cdot n$. Hence, for reactive PLTSs the complexity result of Theorem 4 can be rewritten as follows. (More precisely, we reformulate the result stating that $\mathcal{O}\left(mn + n^2 + \sum_\alpha m_\alpha^2\right)$ and $\mathcal{O}\left((mn^6 + \sum_\alpha m_\alpha^2 n^3)/\log n\right)$ are upper bounds for the space and time complexity of the simulation problem in PLTSs.)

Theorem 5 *The simulation preorder of a reactive PLTS (with n states and the action set Act) can be computed in time $\mathcal{O}(|Act|n^7/\log n)$ and space $\mathcal{O}(|Act|n^2)$.*

Some implementation details: We now describe some more details of an implementation that yields the complexity stated in Theorem 4. For the given PLTS, we use a representation as suggested for the bisimulation algorithm. That is, we fix an orderings $s_1, \ldots, s_n$ for the states and $\alpha_1, \ldots, \alpha_k$ for the actions and deal with a two-dimensional array $\texttt{Steps}[i, j]$ whose elements are pointers to lists that connect (pointers to) the distributions $\mu \in Steps_{\alpha_i}(s_j)$. For the distributions, a representation by a real array $\langle \mu(s_1), \ldots, \mu(s_n) \rangle$ is sufficient.

We represent the set $Sim_{(s,\alpha,\mu)}(s')$ as a list consisting of pointers to the distributions $\mu \in Steps_\alpha(s')$. For these lists $Sim_{(s,\alpha,\mu)}(s')$, we use the operations $First(\cdot)$ which yields the first element of $(\cdot)$ and $Next(\cdot)$ which removes the first element of $(\cdot)$, i.e. the list pointer is shifted to the second element. (The operation $Next(\cdot)$ is only defined for nonempty lists. For a list L consisting of a single element, $Next(L)$ yields the empty list.) The initial ordering of the distributions in $Sim_{(s,\alpha,\mu)}(s')$ is arbitrary. As soon as $\mu \not\sqsubseteq_R \mu'$ is detected for $\mu' = First(Sim_{(s,\alpha,\mu)}(s'))$ we apply the operator $Next(Sim_{(s,\alpha,\mu)}(s'))$ (which corresponds to the removement of μ' from $Sim_{(s,\alpha,\mu)}(s')$).

We organize the current relation R as a queue where the ordering in the initial queue is arbitrary. To avoid the use of the excplicit use of the second relation R_{old} (as it is indicated in the basic algorithm), we use a special variable *last* which is either undefined (i.e. $last = \bot$) or an element (t_0, t_0') of R. Initially, *last* is the last element of R. In the case where $last = (t_0, t_0')$ then none pair (t, t') in R that has been investigated after the last investigation of (t_0, t_0') was removed from R. (Here, by an *investigation* of an element (s, s') of R, we mean the test whether $s \sqsubseteq_R s'$.) I.e. we have $t \sqsubseteq_R t'$ for all pairs (t, t') that are behind (on the right of) of (t_0, t_0') in R (see Figure 19). We use the usual operators

$$R: \quad \underbrace{(s,s')}_{Front(R)} \quad \ldots \quad \underbrace{\underbrace{(t_0,t_0')}_{=last} \quad \ldots \quad (t,t') \quad \ldots}_{t \sqsubseteq_R t'}$$

Figure 19: R organized as a queue

$Front(R)$ which returns and removes the first element of R (under the assumption that R is not empty) and $Add(R, (s, s'))$ which adds the pair (s, s') at the end of R. The choice of an element $(s, s') \in R$ that we investigate next (i.e. for which we check whether $s \sqsubseteq_R s'$) now means that we take the first element of R (via the operation $Front(R)$).

The reformulation of our basic algorithm according to these data structures is shown in Figure 20. (In the initialization step, we use the assignment $R := \emptyset$ to denote that we create an empty queue. For the output, we identify R with the set of pairs contained in R.) We use auxiliary boolean variables *sim*, *dsim* and *done*. While the meaning of *sim* is as in the basic algorithm (i.e. it stands for the condition $s \sqsubseteq_R s'$) the truth value of *dsim* is given by

$$dsim \;=\; \begin{cases} true & : \text{ if } \mu \sqsubseteq_R \mu' \\ false & : \text{ otherwise} \end{cases}$$

where $s \xrightarrow{\alpha} \mu$ is the transition for which we search a matching transition $s' \xrightarrow{\alpha} \mu'$ and where $\mu' = First\left(Sim_{(s,\alpha,\mu)}(s')\right)$ is the current candidate that we investigate. The boolean variable *done* tells us when the relation R agrees with the simulation preorder (up to the "trivial" pairs (s, s)), i.e. when the algorithm should halt.

When we have verified the condition $s \sqsubseteq_R s'$ for the current pair (s, s') (i.e. the boolean variable *sim* is true) then we reinsert (s, s') into R (via the operation $Add(R, (s, s'))$) and distinguish two cases:

- If $last = (s, s')$ then $t \sqsubseteq_R t'$ for all elements (t, t') in R. Then, R agrees with the simulation preorder $\sqsubseteq$. (Thus, the variable *done* becomes true.)
- If $(s, s') \neq last$ then we are in the situation of Figure 19 where $last = (t_0, t_0') \neq (s, s')$. In this case, we continue to investigate the next pair in R but still deal with $last = (t_0, t_0')$.

If $s \not\sqsubseteq_R s'$ then we put $last = \bot$ and "remove"(i.e. do not reinsert) the current pair (s, s') from R. (Note that the removement of (s, s') from R is already part of the operator $Front(R)$.) In this case, the relation $\sqsubseteq_R$ on S might have changed. Thus, $t \sqsubseteq_R t'$ might be violated for some pair in R. For this reason, *last* is undefined as long as we "wait"

Initialization:

- $R := \emptyset$; $last := \bot$; $done := false$;
- For all $(s, s') \in S \times S$ with $s \neq s'$ and $act(s) \subseteq act(s')$ do
  - $Add(R, (s, s'))$;
  - If $last = \bot$ then $last := (s, s')$;
  - For all $\alpha \in Act$ and $\mu \in Steps_\alpha(s)$ do $Sim_{(s,\alpha,\mu)}(s') := Steps_\alpha(s')$;

Iteration: While $R = \emptyset$ and $\neg done$ do

- $(s, s') := Front(R)$; $sim := true$;
- For all $\alpha \in Act$ and $\mu \in Steps_\alpha(s)$ do
  - Repeat
    $\mu' := First\left(Sim_{(s,\alpha,\mu)}(s')\right)$;
    If $\mu \sqsubseteq_R \mu'$ then $dsim := true$
    else begin $dsim := false$; $Next\left(Sim_{(s,\alpha,\mu)}(s')\right)$ end;
  - until $dsim$ or $Sim_{(s,\alpha,\mu)}(s') = \emptyset$;
  - If $\neg dsim$ then $sim := false$;
- If sim then
  - $Add(R, (s, s'))$;
  - If $last = (s, s')$ then $done := true$;
  - If $last = \bot$ then $last := (s, s')$;

  else $last := \bot$;

Output: Return $R \cup \{(s, s) : s \in S\}$.

Figure 20: Algorithm for computing the simulation preorder in a PLTS

for the next pair (t_0, t_0') that will be not removed from R. Thus, when we investigate (s, s') where $last = \bot$ (i.e. the algorithm is in "waiting mode" in the above sense) and get $s \sqsubseteq_R s'$ then we put $last = (s, s')$.

Example 5.6 We apply our algorithm for computing the simulation preorder to the PLTS shown in Figure 21. In the initialization step, we obtain the queue R containing the pairs

- (s_i, s_j), (s_i, v), (v, s_i), (v, v'), $i, j = 1, 2$, $i \neq j$, v, $v' \in \{v_1, v_2, w\}$, $v \neq v'$
- (u, u'), u, $u' \in U = \{u_1^k, u_2^h : k = 0, \ldots, 4, h = 0, 1, 2\}$, $u \neq u'$
- (t_1, t_2), (t_2, t_1)
- (u, t_1), (u, s_i), (u, v), $u \in U$, $v \in \{v_1, v_2, w\}$, $i = 1, 2$.

The pairs (s_2, s_1), (s_i, w), (w, v_j), (s_i, v_j), i, $j = 1, 2$, are removed during their first investigation. For instance, for the pair (w, v_1) the algorithm computes the maximum flow of the network

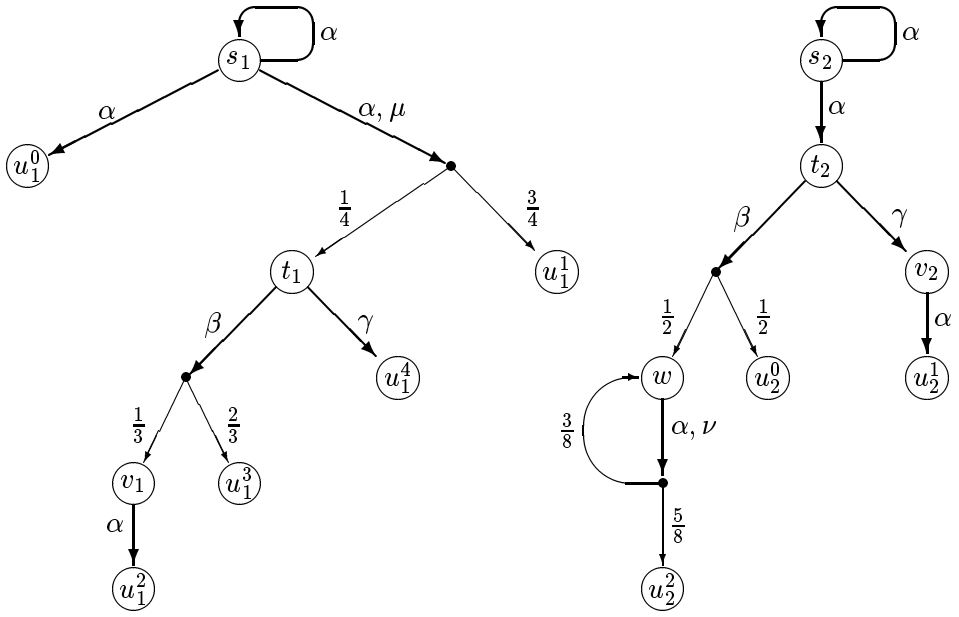
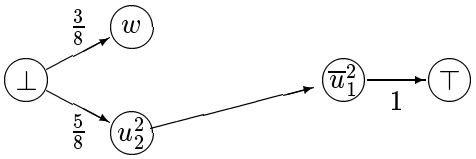


Figure 21:



which is 5/8. Thus, there is no transition of v_1 that can "simulate" the transition $w \xrightarrow{\alpha} \nu$. For the pair (s_2, s_1) the algorithm tries to find an outgoing transition from s_1 which can "simulate" the transition $s_2 \xrightarrow{\alpha} t_2$. As $\rho \not\sqsubseteq_R \mu_{t_2}^1$ for all $\rho \in Steps_\alpha(s_1) = \{\mu_{u_1^0}^1, \mu_{s_1}^1, \mu\}$ the pair (s_2, s_1) is removed from R. The first investigation of (s_1, s_2) yields

$$Sim_{(s_1,\alpha,\mu_{u_1^0}^1)}(s_2) \;=\; \{\mu_{t_2}^1, \mu_{s_2}^1\}$$

and $Sim_{(s_1,\alpha,\mu_{s_1}^1)}(s_2) \;=\; \{\mu_{s_2}^1\}$, $Sim_{(s_1,\alpha,\mu)}(s_2) \;=\; \{\mu_{t_2}^1\}$ (as $\mu \not\sqsubseteq_R \mu_{s_2}^1$ and $\mu_{s_1}^1 \not\sqsubseteq_R \mu_{t_2}^1$).

We suppose that initially the pair (t_2, t_1) is the last element of R (i.e. $last = (t_2, t_1)$ after the initialization step). Then, the first investigation of (t_2, t_1) yields $t_2 \not\sqsubseteq_R t_1$ (as (w, v_1) is already removed from R). After the removement of (t_2, t_1) we have $x \sqsubseteq_R y$ for all $(x, y) \in R$. Hence, if (x, y) is the element of R which is investigated immediately after the removement of (t_2, t_1) then the algorithm sets $last = (x, y)$ after the (second) investigation of (x, y). After investigating all remaining elements of R once more, we the pair $(x, y) = last$ is again at the front of R. Thus, when investigating (x, y) for the third time we put $done = true$ and leave the while-loop. The algorithm returns the simulation preorder which consists of the following pairs.

- (s_1, s_2), (t_1, t_2)
- (v_i, s_j), (w, s_j), (v_i, w), $i,\ j = 1, 2$
- (u, s), $u \in U$, $s \in \{s_1, s_2, t_1, t_2, v_1, v_2, w\}$

and all pairs (s, s), $s \in S$. ■

6 Fully probabilistic systems

So far, we considered probabilistic systems with non-determinism (modelled by PLTSs) where the choice which transition is performed is beyond the control of the system and cannot be supposed to obey certain probabilistic assumptions. For several applications, e.g. for specifying the behaviour of a sequential randomized process [5] or parallel systems whose components are sequential probabilistic processes that work synchronously [29, 30, 45, 33, 6, 34] or concurrent systems with a probabilistic merge operator [2, 24], it suffices to consider simpler models based on Markov chains. Such models, often called *fully probabilistic* or *generative* models, allow for probabilistic choice but not for non-determinism.

In this section, we consider bisimulation and simulation-like relations on a fully probabilistic variant of LTSs. Intuitively, fully probabilistic LTSs (abbrev. FPLTSs) arise from ordinary LTSs by augmenting the outgoing transitions of a state s with probabilities that sum up to (at most) one. Thus, a fully probabilistic LTS can be viewed as an LTS $(S, Act, \rightarrow)$ together with a transition probability function $\mathbf{P}$ where $\mathbf{P}(s, \alpha, t)$ specifies the probability for the event that the transition $s \stackrel{\alpha}{\rightarrow} t$ is the chosen one provided that the system is in state s. For example, the use of a probabilistic merge operator for modelling concurrent systems by a FPLTSs [2, 24] is motivated by the assumption that the probabilities for the possible outcomes of the decisions of a scheduler (who decides which subprocess performs the next step) are known. In this case, the transition probabilities $\mathbf{P}(s, \alpha, t)$ are derived from the probabilistic assumptions about the scheduler. For modelling synchronous probabilistic processes by FPLTSs, one assumes that each step of the sunchronous parallel composition $\mathcal{P}_1 \times \mathcal{P}_2$ is composed by the simultanous execution of transitions of both components $\mathcal{P}_1$ and $\mathcal{P}_2$ where the probabilistic choices in $\mathcal{P}_1$ and $\mathcal{P}_2$ are resolved independently. Thus, the transition probabilities in the FPLTS for $\mathcal{P}_1 \times \mathcal{P}_2$ are obtained by multiplying the transition probabilities for the individual moves of $\mathcal{P}_1$ and $\mathcal{P}_2$. (For further details see e.g. [29, 30, 45, 5].)

Definition 6.1 A *fully probabilistic labelled transition system* (FPLTS for short) is a tuple $(S, Act, \mathbf{P})$ consisting of a finite set S of states and a *transition probability function*

$$\mathbf{P} : S \times Act \times S \ \rightarrow \ [0, 1]$$

such that

$$\sum_{\alpha \in Act} \sum_{t \in S} \mathbf{P}(s, \alpha, t) \ \in \ \{0, 1\}$$

for all $s \in S$. For $C \subseteq S$, we put $\mathbf{P}(s, \alpha, C) = \sum_{t \in C} \ \mathbf{P}(s, \alpha, t)$. A state $s \in S$ is said to be *terminal* iff $\sum_{\alpha, t} \ \mathbf{P}(s, \alpha, t) = 0$. ■

Bisimulation equivalence reformulated for a FPLTS $(S, Act, \mathbf{P})$ [42, 30] is the coarsest equivalence R on the state space S such that $\mathbf{P}(s, \alpha, C) = \mathbf{P}(s', \alpha, C)$ for all $(s, s') \in R$, $\alpha \in Act$ and $C \in S/R$. The bisimulation equivalence classes of a FPLTS can be computed with the splitter/partitioning technique that we sketched in Section 4 for reactive PLTSs (see Figure 6). We recall the complexity result stated by Huynh & Tian [39].

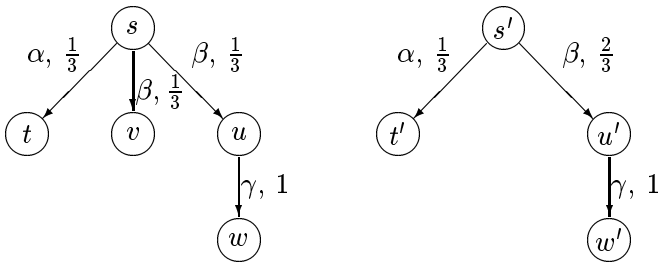


Figure 22: $s \sqsubseteq s'$

Theorem 6 (cf. [39]) *Given a FPLTS $(S, Act, \mathbf{P})$ with n states and k tuples (s, α, t) with $\mathbf{P}(s, \alpha, t) > 0$, the bisimulation equivalence classes can be computed in time $\mathcal{O}(k \log n)$ and space $\mathcal{O}(k + n)$.*

We adapt the definition of the simulation preorder à la Segala & Lynch [57] (Definition 3.6) for FPLTSs. Let $(S, Act, \mathbf{P})$ be a FPLTS, s, $s' \in S$ and $R \subseteq S \times S$. If s is non-terminal then a *weight function* for (s, s') w.r.t. R is a function $\delta : S \times Act \times S \to [0, 1]$ such that for all $\alpha \in Act$ and t, $t' \in S$:

1. If $\delta(t, \alpha, t') > 0$ then $(t, t') \in R$.
2. $$\sum_{u' \in S} \delta(t, \alpha, u') \;=\; \mathbf{P}(s, \alpha, t), \qquad \sum_{u \in S} \delta(u, \alpha, t') \;=\; \mathbf{P}(s', \alpha, t').$$

We write $s \sqsubseteq_R s'$ iff either s is terminal or there exists a weight function for (s, s') w.r.t. R. A *simulation* for (S, P) is a binary relation R on S such that $s \sqsubseteq_R s'$ for all $(s, s') \in R$. We say s *implements* s' and s' *simulates* s (denoted by $s \sqsubseteq s'$) iff there exists a simulation that contains (s, s').

Example 6.2 Consider the FPLTS of Figure 22. The relation

$$R \;=\; \{\; (s, s'),\; (t, t'),\; (u, u'),\; (v, u'),\; (w, w') \;\}$$

is a simulation as e.g. $\delta(t, \alpha, t') \;=\; \delta(v, \beta, u') \;=\; \delta(u, \beta, u') \;=\; 1/3$ (and $\delta(\cdot) = 0$ in all other cases) is a weight function for (s, s') w.r.t. R. Hence, $s \sqsubseteq s'$. ■

As in the case of reactive PLTSs, in FPLTSs bisimulation equivalence coincides with simulation equivalence (which is defined as the kernel of the simulation preorder) [5]. Thus, the complexity result stated by [39] (Theorem 6) also holds for simulation equivalence.

Theorem 7 *The simulation equivalence classes in a FPLTS can be computed in time $\mathcal{O}(k \log n)$ and space $\mathcal{O}(k + n)$. (Here, n and k are as in Theorem 6.)*

We now briefly explain how the network-based technique that we suggested in Section 5 can be modified for computing the simulation preorder in FPLTSs. In what follows, we fix a FPLTS $(S, Act, \mathbf{P})$. For s, $s' \in S$ and $R \subseteq S \times S$ we define $\mathcal{N}(s, s', R)$ to be the network (N, E, c) where

$$N \;=\; \{\bot,\top\} \cup Act \times (S \cup \overline{S}),\ \overline{S} = \{\overline{t} : t \in S\}$$
$$E \;=\; \{(\bot, \langle\alpha,t\rangle), (\langle\alpha,\overline{t}\rangle,\top) : t \in S, \alpha \in Act\} \;\cup\; \{(\langle\alpha,t\rangle, \langle\alpha,\overline{u}\rangle) : (t,u) \in R\}$$
$$c(\bot,\langle\alpha,t\rangle) \;=\; \mathbf{P}(s,\alpha,t),\ c(\langle\alpha,\overline{t}\rangle,\top) \;=\; \mathbf{P}(s',\alpha,t), \quad c(\langle\alpha,t\rangle,\langle\alpha,\overline{u}\rangle) \;=\; 1.$$

Similar to Lemma 5.1 it can be shown that $s \sqsubseteq_R s'$ iff either s is terminal or the maximum flow in $\mathcal{N}(s,s',R)$ is 1. Thus, the simulation preorder of a FPLTS can be computed by the following method. We start with the preorder

$$R \;=\; \{(s,s') \in S \times S' : \text{if } s' \text{ is terminal then } s \text{ is terminal}\}.$$

As long as there is a pair $(s,s') \in R$ where s is non-terminal and $s \not\sqsubseteq_R s'$ (what we test by computing the maximum flow in $\mathcal{N}(s,s',R)$) we remove (s,s') from R. This method can be implemented similar to the one proposed in section 5 (Figure 20). The time complexity is as in the reactive case. We obtain:

Theorem 8 *The simulation preorder of a FPLTS (with n states and the action set Act) can be computed in time $\mathcal{O}(|Act|n^7/\log n)$ and space $\mathcal{O}(|Act|n^2)$.*

In many applications, one wants only to give *lower* and *upper bounds* for the probabilities of an acceptable system behaviour rather than the exact probabilities. Jonsson & Larsen [40] deal with fully probabilistic systems where the states are labelled by atomic propositions and define a notion of "satisfaction relation" that relates the states of a given fully probabilistic system and the states of a *fully probabilistic specification system* which prescribes intervals of allowed probabilities. We now adapt the definitions given in [40] for our setting (i.e. for systems with action labels for the transitions).

Definition 6.3 (cf. [40]) A *fully probabilistic specification system* (FPSS) is a tuple $(\overline{S}, Act, \overline{\mathbf{P}})$ where $\overline{S}$ is a finite set of states and $\overline{\mathbf{P}} : \overline{S} \times Act \times \overline{S} \to 2^{[0,1]}$ is a function such that, for all $\overline{s}$, $\overline{t} \in \overline{S}$ and $\alpha \in Act$, $\overline{\mathbf{P}}(\overline{s},\alpha,\overline{t})$ is a closed interval contained in $[0,1]$. Let $(S, Act, \mathbf{P})$ be a FPLTS. If $R \subseteq S \times \overline{S}$ and $s \in S$, $\overline{s} \in \overline{S}$ then $s\ sat_R\ \overline{s}$ iff either s is terminal or there exsits a *weight function* for $(s,\overline{s})$ w.r.t. R, i.e. a function

$$\delta : S \times Act \times \overline{S} \to [0,1]$$

such that for all $\alpha \in Act$ and $t \in S$, $\overline{t} \in \overline{S}$:

1. If $\delta(t,\alpha,\overline{t}) > 0$ then $(t,\overline{t}) \in R$.
2. $$\sum_{\overline{u} \in \overline{S}} \delta(t,\alpha,\overline{u}) \;=\; \mathbf{P}(s,\alpha,t), \qquad \sum_{u \in S} \delta(u,\alpha,\overline{t}) \;\in\; \overline{\mathbf{P}}(\overline{s},\alpha,\overline{t}).$$

A *satisfaction relation* for a FPLTS $(S, Act, \mathbf{P})$ and a FPSS $(\overline{S}, Act, \overline{\mathbf{P}})$ is a binary relation $R \subseteq S \times \overline{S}$ such that $s\ sat_R\ \overline{s}$ for all $(s,\overline{s}) \in R$. We write $s\ sat\ s'$ iff $(s,\overline{s})$ is contained in some satisfaction relation for $(S, Act, \mathbf{P})$ and $(\overline{S}, Act, \overline{\mathbf{P}})$. ■

The relation $sat \subseteq S \times \overline{S}$ can be computed similar to the way in which we compute the simulation preorder of a FPLTS; the only difference being the use of networks with lower and upper bounds, see e.g. [26]. We start with the relation $R = S \times \overline{S}$ and successively

remove those pairs $(s, \overline{s})$ from R where $\neg(s\ sat_R\ \overline{s})$. For the test whether $s\ sat_R\ \overline{s}$ we compute the maximum flow in the network $\mathcal{N}(s, \overline{s}, R) = (N, E, c_l, c_u)$ where

$$\begin{array}{rcl} N & = & \{\bot, \top\} \cup Act \times (S \cup \overline{S}) \\ E & = & \{(\bot, \langle \alpha, t\rangle), (\langle \alpha, \overline{u}\rangle, \top) : t \in S, \alpha \in Act, \overline{u} \in \overline{S}\} \\ & & \cup\ \{(\langle \alpha, t\rangle, \langle \alpha, \overline{u}\rangle) : (t, \overline{u}) \in R\} \end{array}$$

$$c_l(\bot, <\alpha, t>) = c_l(\langle \alpha, t\rangle, \langle \alpha, \overline{u}\rangle) = 0, \quad c_l(\langle \alpha, \overline{t}\rangle, \top) = \min \overline{\mathbf{P}}(\overline{s}, \alpha, \overline{t})$$

$$c_u(\bot, \langle \alpha, t\rangle) = \mathbf{P}(s, \alpha, t), \quad c_u(\langle \alpha, \overline{t}\rangle, \top) = \max \overline{P}(\overline{s}, \alpha, \overline{t}),$$

$$c_u(\langle \alpha, t\rangle, \langle \alpha, \overline{u}\rangle) = 1.$$

Here, c_l, c_u are functions that assign to each edge $e \in E$ the lower bound $c_l(e)$ and upper bound $c_u(e)$ of the possible flow through e. Similarly to Lemma 5.1, $s\ sat_R\ \overline{s}$ iff either s is terminal or the maximum flow in $\mathcal{N}(s, \overline{s}, R)$ is 1. The problem of finding the maximum flow in a network with lower and upper bounds can be reduced to the computation of the maximum flow in a "usual" network of the same asymptotic size (see e.g. [26]). Hence:

Theorem 9 *The satisfaction relation $sat \subseteq S \times \overline{S}$ for a FPLTS $(S, Act, \mathbf{P})$ and a FPSS $(\overline{S}, Act, \overline{\mathbf{P}})$ can be computed in time $\mathcal{O}(|Act|(n+\overline{n})^7 / \log(n+\overline{n}))$ and space $\mathcal{O}(|Act|(n+\overline{n})^2)$ where $n = |S|$ and $\overline{n} = |\overline{S}|$.*

7 Concluding remarks

We presented algorithms for computing the bisimulation equivalence classes and the simulation preorder of probabilistic systems modelled by probabilistic extensions of ordinary finite LTSs: PLTSs where probabilistic and non-deterministic choice coexist and FPLTSs where all choices are probabilistic. In either case, the worst case time complexity is polynomial in the size of state space and number of transitions.

To the best of our knowledge, no lower bounds for the complexity of bisimulation or simulation in PLTSs or FPLTSs are known. However, we argue that at least our algorithm for bisimulation equivalence in PLTSs (which runs in time $\mathcal{O}(mn(\log m + \log n))$ where n is the number of states and m the number of transitions) is quite efficient and seems to be an adequate basis for a verification tool that generates and analyses the quotient space $S/\sim$ rather than the (possibly much bigger) original system. As mn is an upper bound for the number of edges in the underlying directed graph G of the given PLTS (obtained by ignoring the probabilities), the worst case time bound $\mathcal{O}(mn(\log m + \log n))$ for bisimulation equivalence in PLTSs can be rewritten as $\mathcal{O}(|G| \log |G|)$ where $|G|$ denotes the "size" of the graph G (i.e. the number of nodes and edges in G which is bounded by $n+mn$). Compared with the non-probabilistic case, where the time complexity $\mathcal{O}(m \log n)$ à la Paige & Tarjan [51] reformulated in the size of the underlying directed graph G yields the same upper bound $\mathcal{O}(|G| \log |G|)$, our worst case time complexity seems to be reasonable. In those cases where $\mathcal{O}(\log m) = \mathcal{O}(\log n)$, e.g. if the system arises through the parallel execution of l sequential probabilistic processes for some fixed l, the time complexity for computing the bisimulation equivalence classes is $\mathcal{O}(n^2 \log n)$ which is roughly the same as in FPLTSs or reactive PLTSs [39].

Our algorithm for the simulation preorder has the time complexity $\mathcal{O}((mn^6+m^2n^3)/\log n)$ in PLTSs and $\mathcal{O}(|Act|n^7/\log n)$ in FPLTSs while the simulation preorder in LTSs can be computed in time $\mathcal{O}(mn)$ [38]. Although our informal arguments (given at the beginning of Section 5) demonstrate that a straightforward reformulation of the technique proposed in [38] does not work for the probabilistic setting, several improvements of our methods might be possible to get an algorithm whose worst case time complexity is closer to the $\mathcal{O}(mn)$ time bound à la [38]. In particular, we expect that it should be possible to develop special (more efficient) algorithms for computing the quotient space $S/\sim_{\mathrm{sim}}$ of a PLTSs.

For an implementation, we suggested the use of "conventional" data structures (connected lists, queues, arrays, etc.). In particular, one might wonder why we propose to represent the distributions by real arrays which leads to the asymptotic size $\Theta(n)$ for the representation of any distribution. This was important for the analysis of the time complexity as the use of real arrays makes it possible to calculate the values $\mu(C) = \sum_{t\in C} \mu(t)$ in time $\mathcal{O}(|C|)$. (We used the latter observation to establish the $\mathcal{O}(mn \log n)$ time bound for the splitting operations $\mathit{Split}(\mathcal{M}, C)$ in the bisimulation algorithm. See Lemma 4.8.) On the other hand, when one focusses on a higher priority for space-efficiency than time-efficiency then other data structures for the representation of the distributions might be preferable; in particular, in those cases where $|\mathit{Supp}(\mu)| = |\{s \in S : \mu(s) > 0\}|$ is much smaller than $n = |S|$. For instance, the use of *multi-terminal* (or *algebraic*) *binary decision diagrams* [19, 3], briefly MTBDDs, seems to be very promising as MTBDDs are known to be a space-efficient data structure for verifying probabilistic systems against temporal logical specifications [7, 33, 34]. The use of MTBDDs relies on a *symbolic* representation of the state space (where sets of states are represented rather than the states themselves). In [5], the theoretical foundations for a MTBDD-based algorithm that computes the bisimulation equivalence classes of a FPLTS or reactive PLTS are given. The proposed technique uses a greatest fixed point characterization of bisimulation equivalence and is in the spirit of [14, 25] where a BDD-based model checker for the relational mu-calculus serves as basis for a symbolic method that computes the bisimulation equivalence classes in ordinary LTSs. However, it is not yet clear how an efficient MTBDD-based representation of general PLTSs can be obtained; let alone how to treat bisimulation or simulation for general PLTSs with MTBDDs. Unfortunately, we cannot report on experimental results, neither for the MTBDD-based approach for FPLTSs or reactive PLTSs nor for the algorithms presented in this paper. An implementation and case studies will be topics of future work.

In this paper, we only considered *strong* (bi-)simulation which does not abstract from internal actions. Notions of *weak bisimulations* on probabilistic variants of LTSs (i.e. branching time equivalences that are insensitive with respect to certain internal transitions) have been defined in [57, 8, 53]. In [8], an $\mathcal{O}(n^3)$ algorithm for deciding weak bisimilarity in FPLTSs is proposed. In the forthcoming papers [53, 10], weak equivalences on (variants of) PLTSs are proposed and shown to be decidable in polynomial time. However, as far as the authors know, the complexity (or even decidability) of weak (bi-)simulation for PLTS à la Segala & Lynch [57] is still an open problem.

References

[1] G. Adel'son-Velshii, Y. Landis: An Algorithm for the Organization of Information, Soviet. Math. Dokl. 3, pp 1259-1262, 1962.

[2] J. Baeten, J. Bergstra, S. Smolka: Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, Proc. CONCUR'92, *Lecture Notes in Computer Science*, Vol. 630, pp 472-485, 1992. The full version with the same title has appeared in *Information and Computation*, Vol. 122, pp 234-255, 1995.

[3] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Padro, F. Somenzi: Algebraic Decision Diagrams and their Applications, Proc. ICCAD'93, pp 188-191, 1993. The full version with the same title has appeared in *Formal Methods in Systems Design*, Vol. 10, No. 2/3, pp 171-206, 1997.

[4] C. Baier: Polynomial Time Algorithms for Testing probabilistic Bisimulation and Simulation, Proc. CAV'96, *Lecture Notes in Computer Science*, Vol. 1102, pp 38-49, 1996.

[5] C. Baier: On Algorithmic Verification Methods for Probabilistic Systems, Habilitation Thesis, Universität Mannheim, 1998.

[6] C. Baier, E. Clarke, V. Hartonas-Garmhausen: On the Semantical Foundations of Probabilistic Synchronous Reactive Programs, Proc. PROBMIV'98, *Electronic Notes in Theoretical Computer Science*, Vol. 22, 1999.

[7] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, M. Ryan: Symbolic Model Checking for Probabilistic Processes, Proc. ICALP'97, *Lecture Notes in Computer Science*, Vol. 1256, pp 430-440, 1997.

[8] C. Baier, H. Hermanns: Weak Bisimulation for Fully Probabilistic Processes, Proc. CAV'97, *Lecture Notes in Computer Science*, Vol. 1254, pp 119-130, 1997.

[9] C. Baier, M. Kwiatkowska: Domain Equations for Probabilistic Processes, to appear in *Mathematical Structurs in Computer Science*, 1999. Prelimary version in Proc. EXPRESS'97, *Electronic Notes in Theoretical Computer Science*, Vol. 7, 1997.

[10] C. Baier, M. Stoelinga: Abstraction in Probabilistic Systems with Normed Bisimulation, in preparation, 1999.

[11] A. Bianco, L. de Alfaro: Model Checking of Probabilistic and Nondeterministic Systems, Proc. Foundations of Software Technology and Theoretical Computer Science, *Lecture Notes in Computer Science*, Vol. 1026, pp 499-513, 1995.

[12] T. Bolognesi, S. Smolka: Fundamental Results for the Verification of Observational Equivalence: a Survey, *in* "Protocol Specification, Testing and Verification", Elsevier Science Publishers, IFIP, pp 165-179, 1987.

[13] S. Brookes, C.A.R. Hoare, A. Roscoe: A Theory of Communicating Sequential Processes, *Journal of the ACM*, Vol. 31 (3), pp 560-599, 1984.

[14] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang: Symbolic Model Checking: 10^{20} States and Beyond, Proc. LICS'90, pp 428-439, 1990. The full version with the same title has appeared in *Information and Computation*, Vol. 98 (2), pp 142-170, 1992.

[15] J. Cheriyan, T. Hagerup, K. Mehlhorn: Can a Maximum Flow be Computed in $\mathcal{O}(nm)$ Time?, Proc. ICALP'90, *Lecture Notes in Computer Science*, Vol. 443, pp 235-248, 1990.

[16] I. Christoff: Testing Equivalences and Fully Abstract Models for Probabilistic Processes, Proc. CONCUR'92, *Lecture Notes in Computer Science*, Vol. 458, pp 126-140, 1990.

[17] L. Christoff, I. Christoff: Efficient Algorithms for Verification of Equivalences for Probabilistic Processes, Proc. CAV'91, *Lecture Notes in Computer Science*, Vol. 575, pp 310-321, 1991.

[18] L. Christoff: Specification and Verification Methods for Probabilistic Processes, Ph. D. Thesis, Department of Computer Science, Uppsala University, 1993.

[19] E. Clarke, M. Fujita, P. McGeer, J. Yang, X. Zhao: Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation, In Proc. IWLS'93: International Workshop on Logic Synthesis, Tahoe City, 1993.

[20] R. Cleaveland, J. Parrow, B. Steffen: A Semantic Based Verification Tool for Finite State Systems, *in* "Protocol Specification, Testing and Verification IX", Elsevier Science Publishers, IFIP, pp 287-302, 1990.

[21] R. Cleaveland, S. Smolka, A. Zwarico: Testing Preorders for Probabilistic Processes, Proc. ICALP 1992, *Lecture Notes in Computer Science*, Vol. 623, pp 708-719, 1992.

[22] E. Dinic: Algorithm for Solution of a Problem of Maximal Flow in a Network with Power Estimation, *Soviet. Math. Dokl.*, Vol. 11, pp 1277-1280, 1970.

[23] L. de Alfaro: "Formal Verification of Probabilistic Systems", Ph.D.Thesis, Stanford University, 1997.

[24] P. d'Argenio, H. Hermanns, J. Katoen: On Generative Parallel Composition, Proc. PROBMIV'98, *Electronic Notes in Theoretical Computer Science*, Vol. 22, 1999.

[25] R. Enders, T. Filkorn, D. Taubner: Generating BDDs for Symbolic Model checking in CCS, *Distributed Computing*, Vol. 6, pp 155-164, 1993.

[26] S. Even: "Graph Algorithms", Computer Science Press, 1979.

[27] J.C. Fernandez: An Implementation of an Efficient Algorithm for Bisimulation Equivalence, *Science of Computer Programming*, Vol. 13, pp 219-236, 1989/90.

[28] L. Ford, D. Fulkerson: "Flows in Networks", Princeton University Press, 1962.

[29] A. Giacalone, C. Jou, S. Smolka: Algebraic Reasoning for Probabilistic Concurrent Systems, *in* Proc. IFIP TC2 Working Conference on Programming Concepts and Methods, 1990.

[30] R. van Glabbeek, S. Smolka, B. Steffen, C. Tofts: Reactive, Generative, and Stratified Models for Probabilistic Processes, *in* Proc. LICS'90, pp 130-141, 1990.

[31] J. Groote, F. Vaandrager: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence, Proc. ICALP'90, *Lecture Notes in Computer Science*, Vol. 443, pp 626-638, 1990.

[32] S. Hart, M. Sharir, A. Pnueli: Termination of Probabilistic Concurrent Programs, *ACM Transactions on Programming Languages*, Vol. 5, pp 356-380, 1983.

[33] V. Hartonas-Garmhausen: Probabilistic Symbolic Model Checking with Engineering Models and Applications, Ph.D.Thesis, Carnegie Mellon University, 1998.

[34] V. Hartonas-Garmhausen, S. Campos, E. Clarke: ProbVerus: Probabilistic Symbolic Model Checking, Proc. ARTS'99, *Lecture Notes in Computer Science*, Vol. 1601, pp 96-110, 1999.

[35] H. Hansson: Time and Probability in Formal Design of Distributed Systems, *in* "Real-Time Safety-Critical Systems", Vol. 1, Elsevier, 1994.

[36] H. Hansson, B. Jonsson: A Calculus for Communicating Systems with Time and Probabilities, *in* Proc. IEEE Real-Time Systems Symposium, 1990.

[37] H. Hansson, B. Jonsson: A Logic for Reasoning about Time and Probability, *Formal Aspects of Computing*, Vol. 6, pp 512-535, 1994.

[38] M. Henzinger, T. Henzinger, P. Kopke: Computing Simulations on Finite and Infinite Graphs, *in* Proc. FOCS'95, pp 453-462, 1995.

[39] T. Huynh, L. Tian: On some Equivalence Relations for Probabilistic Processes, *Fundamenta Informaticae*, Vol. 17, pp 211-234, 1992.

[40] B. Jonsson, K.G. Larsen: Specification and Refinement of Probabilistic Processes, *in* Proc. LICS'91, pp 266-277, 1991.

[41] B. Jonsson, W. Yi: Compositional Testing Preorders for Probabilistic Processes, *in* Proc. LICS'95, pp 431-443, 1995.

[42] C.C. Jou, S. Smolka: Equivalences, Congruences and Complete Axiomatizations for Probabilistic Processes Proc. CONCUR'90, *Lecture Notes in Computer Science*, Vol. 458, pp 367-383, 1990.

[43] P. Kannelakis, S. Smolka: CCS Expressions, Finite State Processes and Three Problems of Equivalence, *in* Proc. 2nd ACM Symposium on the Principles of Distributed Computing, pp 228-240, 1983. The full version appeared in *Information and Computation*, Vol. 86, pp 43-68, 1990.

[44] K. Larsen, A. Skou: Bisimulation through Probabilistic Testing, Proc. POPL'89, 1989. The full version with the same title has appeared in *Information and Computation*, Vol. 94, pp 1-28, 1991.

[45] K. Larsen, A. Skou: Compositional Verification of Probabilistic Processes, Proc. CONCUR'92, *Lecture Notes in Computer Science*, Vol. 630, pp 456-471, 1992.

[46] V. Malhotra, M. Pramodh Kumar, S. Maheshwari: An $\mathcal{O}(|V^3|)$ Algorithm for Finding Maximum Flows in Networks, *Computer Science Program*, Indian Institute of Technology, Kanpur 208016, 1978.

[47] R. Milner: A Calculus of Communicating Systems, *Lecture Notes in Computer Science*, Vol. 92, 1980.

[48] R. Milner: "Communication and Concurrency", Prentice Hall, 1989.

[49] I. Nievergelt, E. Reinhold: Binary Search Trees of Bounded Balance, *SICOMP* 2, pp 33-43, 1973.

[50] R. de Nicola, M. Hennessy: Testing Equivalences for Processes, *Theoretical Computer Science*, Vol. 34, pp 83-133, 1983.

[51] R. Paige, R. Tarjan: Three Partition Refinement Algorithms, SIAM Journal of Computing, Vol. 16, No. 6, pp 973-989, 1987.

[52] D. Park: Concurrency and Automata on Infinite Sequences, Proc. 5th GI Conference, *Lecture Notes in Computer Science*, Vol. 104, pp 167-183, 1981.

[53] A. Philippou, O. Sokolsky, I. Lee: Weak Bisimulation for Probabilistic Systems, submitted for publication.

[54] A. Pnueli, L. Zuck: Verification of Multiprocess Probabilistic Protocols, *Distributed Computing*, Vol. 1, No. 1, pp 53-72, 1986.

[55] A. Pogosyants, R. Segala, N. Lynch: Verification of the Randomized Consensus Algorithm of Aspnes and Herlihy: a Case Study, Proc. WDAG'97, *Lecture Notes in Computer Science*, Vol. 1320, pp 111-125, 1997.

[56] R. Segala: Modeling and Verification of Randomized Distributed Real-Time Systems, Ph.D. Thesis, Massachusetts Institute of Technology, 1995.

[57] R. Segala, N. Lynch: Probabilistic Simulations for Probabilistic Processes, Proc. CONCUR 94, *Lecture Notes in Computer Science*, Vol. 836, pp 481-496, 1994. The full version has appeared in *Nordic Journal of Computing*, Vol. 2 (2), pp 250-273, 1995.

[58] R. Segala: A Compositional Trace-Based Semantics for Probabilistic Automata, Proc. CONCUR'95, *Lecture Notes in Computer Science*, Vol. 962, pp 234-248, 1995.

[59] M. Stoelinga, F. Vaandrager: Root Contention in IEEE 1394, Proc. ARTS'99, *Lecture Notes in Computer Science*, Vol. 1601, pp 53-74, 1999.

[60] C. Tofts: Processes with Probabilities, *Formal Aspects of Computing*, Vol. 6, No. 5, 1994.

[61] M. Vardi: Automatic Verification of Probabilistic Concurrent Finite-State Programs, *in* Proc. 26th Symp. on Foundations of Computer Science, pp 327-338, 1985.

[62] E. de Vink, J. Rutten: Bisimulation for Probabilistic Transition Systems: A Coalgebraic Approach, Proc. ICALP'97, *Lecture Notes in Computer Science*, Vol. 1256, pp 460-470, 1997.

[63] S. Yuen, R. Cleaveland, Z. Dayar, S. Smolka: Fully Abstract Characterizations of Testing Preorders for Probabilistic Processes, Proc. CONCUR'94, *Lecture Notes in Computer Science*, Vol. 836, pp 497-512, 1994.

[64] W. Yi: Algebraic Reasoning for Real-Time Probabilistic Processes with Uncertain Information, Proc. FTRTFT'94, *Lecture Notes in Computer Science*, Vol. 863, pp 680-693, 1994.

[65] W. Yi, K. Larsen: Testing Probabilistic and Nondeterminsitic Processes, in "Protocol, Specification, Testing and Verification XII", Elsevier Science Publishers, IFIP, pp 47-61, 1992.