

Assume-Guarantee Abstraction Refinement for Probabilistic Systems [★]

Anvesh Komuravelli¹, Corina S. Păsăreanu², and Edmund M. Clarke¹

¹ Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

² Carnegie Mellon Silicon Valley, NASA Ames, Moffett Field, CA, USA

Abstract. We describe an automated technique for assume-guarantee style checking of strong simulation between a system and a specification, both expressed as non-deterministic Labeled Probabilistic Transition Systems (LPTSes). We first characterize counterexamples to strong simulation as *stochastic* trees and show that simpler structures are insufficient. Then, we use these trees in an abstraction refinement algorithm that computes the assumptions for assume-guarantee reasoning as conservative LPTS abstractions of some of the system components. The abstractions are automatically refined based on tree counterexamples obtained from failed simulation checks with the remaining components. We have implemented the algorithms for counterexample generation and assume-guarantee abstraction refinement and report encouraging results.

1 Introduction

Probabilistic systems are increasingly used for the formal modeling and analysis of a wide variety of systems ranging from randomized communication and security protocols to nanoscale computers and biological processes. Probabilistic model checking is an automatic technique for the verification of such systems against formal specifications [2]. However, as in the classical non-probabilistic case [7], it suffers from the *state explosion* problem, where the state space of a concurrent system grows exponentially in the number of its components.

Assume-guarantee style compositional techniques [18] address this problem by decomposing the verification of a system into that of its smaller components and composing back the results, without verifying the whole system directly. When checking individual components, the method uses *assumptions* about the components' environments and then, discharges them on the rest of the system. For a system of two components, such reasoning is captured by the following simple assume-guarantee rule.

$$\frac{1 : L_1 \parallel A \preceq P \quad 2 : L_2 \preceq A}{L_1 \parallel L_2 \preceq P} \text{ (ASYM)}$$

[★] This research was sponsored by DARPA META II, GSRC, NSF, SRC, GM, ONR under contracts FA8650-10C-7079, 1041377 (Princeton University), CNS0926181/CNS0931985, 2005TJ1366, GMCMUCRLNV301, N000141010188, respectively, and the CMU-Portugal Program.

Here L_1 and L_2 are system components, P is a specification to be satisfied by the composite system and A is an assumption on L_1 's environment, to be discharged on L_2 . Several other such rules have been proposed, some of them involving symmetric [19] or circular [8,19,16] reasoning. Despite its simplicity, rule ASYM has been proven the most effective in practice and studied extensively [19,4,11], mostly in the context of non-probabilistic reasoning.

We consider here the *automated* assume-guarantee style compositional verification of *Labeled Probabilistic Transition Systems* (LPTSeS), whose transitions have both probabilistic and non-deterministic behavior. The verification is performed using the rule ASYM where L_1 , L_2 , A and P are LPTSeS and the conformance relation \preceq is instantiated with *strong simulation* [20]. We chose strong simulation for the following reasons. Strong simulation is a decidable, well studied relation between specifications and implementations, both for non-probabilistic [17] and probabilistic [20] systems. A method to help scale such a check is of a natural interest. Furthermore, rule ASYM is both sound and complete for this relation. Completeness is obtained trivially by replacing A with L_2 but is essential for full automation (see Section 5). One can argue that strong simulation is too fine a relation to yield suitably small assumptions. However, previous success in using strong simulation in non-probabilistic compositional verification [5] motivated us to consider it in a probabilistic setting as well. And we shall see that indeed we can obtain small assumptions for the examples we consider while achieving savings in time and memory (see Section 6).

The main challenge in automating assume-guarantee reasoning is to come up with such small assumptions satisfying the premises. In the non-probabilistic case, solutions to this problem have been proposed which use either automata learning techniques [19,4] or abstraction refinement [12] and several improvements and optimizations followed. For probabilistic systems, techniques using automata learning have been proposed. They target *probabilistic reachability* checking and are not guaranteed to terminate due to incompleteness of the assume-guarantee rules [11] or to the undecidability of the conformance relation and learning algorithms used [10].

In this paper we propose a complete, fully automatic framework for the compositional verification of LPTSeS with respect to simulation conformance. One fundamental ingredient of the framework is the use of *counterexamples* (from failed simulation checks) to iteratively refine inferred assumptions. Counterexamples are also extremely useful in general to help with debugging of discovered errors. However, to the best of our knowledge, the notion of a counterexample has not been previously formalized for strong simulation between probabilistic systems. As our first contribution we give a characterization of counterexamples to strong simulation as *stochastic* trees and an algorithm to compute them; we also show that simpler structures are insufficient in general (Section 3).

We then propose an assume-guarantee abstraction-refinement (AGAR) algorithm (Section 5) to automatically build the assumptions used in compositional reasoning. The algorithm follows previous work [12] which, however, was done in a non-probabilistic, trace-based setting. In our approach, A is maintained as a *conservative abstraction* of L_2 , *i.e.* an LPTS that simulates L_2 (hence, premise

2 holds by construction), and is iteratively refined based on tree counterexamples obtained from checking premise 1. The iterative process is guaranteed to terminate, with the number of iterations bounded by the number of states in L_2 . When L_2 itself is composed of multiple components, the second premise ($L_2 \preceq A$) is viewed as a new compositional check, generalizing the approach to $n \geq 2$ components. AGAR can be further applied to the case where the specification P is instantiated with a formula of a logic preserved by strong simulation, such as *safe-pCTL*.

We have implemented the algorithms for counterexample generation and for AGAR using JavaTM and Yices [9] and show experimentally that AGAR can achieve significantly better performance than non-compositional verification.

Other Related Work. Counterexamples to strong simulation have been characterized before as tree-shaped structures for the case of non-probabilistic systems [5] which we generalize to stochastic trees in Section 3 for the probabilistic case. Tree counterexamples have also been used in the context of a compositional framework that uses rule ASYM for checking strong simulation in the non-probabilistic case [4] and employs tree-automata learning to build deterministic assumptions.

AGAR is a variant of the well-known CounterExample Guided Abstraction Refinement (CEGAR) approach [6]. CEGAR has been adapted to probabilistic systems, in the context of probabilistic reachability [13] and *safe-pCTL* [3]. The CEGAR approach we describe in Section 4 is an adaptation of the latter. Both these works consider abstraction refinement in a monolithic, non-compositional setting. On the other hand, AGAR uses counterexamples from checking one component to refine the abstraction of another component.

2 Preliminaries

Labeled Probabilistic Transition Systems. Let S be a non-empty set. $Dist(S)$ is defined to be the set of discrete probability distributions over S . We assume that all the probabilities specified explicitly in a distribution are rationals in $[0, 1]$; there is no unique representation for all real numbers on a computer and floating-point numbers are essentially rationals. For $s \in S$, δ_s is the Dirac distribution on s , i.e. $\delta_s(s) = 1$ and $\delta_s(t) = 0$ for all $t \neq s$. For $\mu \in Dist(S)$, the support of μ , denoted $Supp(\mu)$, is defined to be the set $\{s \in S | \mu(s) > 0\}$ and for $T \subseteq S$, $\mu(T)$ stands for $\sum_{s \in T} \mu(s)$. The models we consider, defined below, have both probabilistic and non-deterministic behavior. Thus, there can be a non-deterministic choice between two probability distributions, even for the same action. Such modeling is mainly used for underspecification and moreover, the abstractions we consider (see Definition 8) naturally have this non-determinism. As we see below, the theory described does not become any simpler by disallowing non-deterministic choice for a given action (Lemmas 4 and 5).

Definition 1 (LPTS). A Labeled Probabilistic Transition System (LPTS) is a tuple $\langle S, s^0, \alpha, \tau \rangle$ where S is a set of states, $s^0 \in S$ is a distinguished start

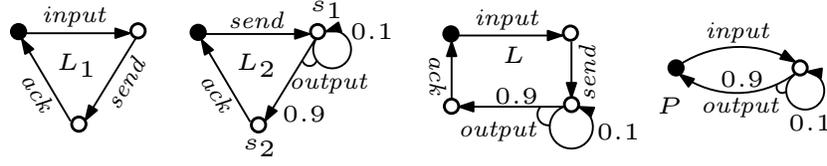


Fig. 1: Four reactive and fully-probabilistic LPTSes.

state, α is a set of actions and $\tau \subseteq S \times \alpha \times \text{Dist}(S)$ is a probabilistic transition relation. For $s \in S$, $a \in \alpha$ and $\mu \in \text{Dist}(S)$, we denote $(s, a, \mu) \in \tau$ by $s \xrightarrow{a} \mu$ and say that s has a transition on a to μ .

An LPTS is called reactive if τ is a partial function from $S \times \alpha$ to $\text{Dist}(S)$ (i.e. at most one transition on a given action from a given state) and fully-probabilistic if τ is a partial function from S to $\alpha \times \text{Dist}(S)$ (i.e. at most one transition from a given state).

Figure 1 illustrates LPTSes. Throughout this paper, we use filled circles to denote start states in the pictorial representations of LPTSes. For the distribution $\mu = \{(s_1, 0.1), (s_2, 0.9)\}$, L_2 in the figure has the transition $s_1 \xrightarrow{\text{output}} \mu$. All the LPTSes in the figure are reactive as no state has more than one transition on a given action. They are also fully-probabilistic as no state has more than one transition. In the literature, an LPTS is also called a *simple probabilistic automaton* [20]. Similarly, a reactive (fully-probabilistic) LPTS is also called a (Labeled) *Markov Decision Process (Markov Chain)*. Also, note that an LPTS with all the distributions restricted to Dirac distributions is the classical (non-probabilistic) *Labeled Transition System (LTS)*; thus a reactive LTS corresponds to the standard notion of a *deterministic LTS*. For example, L_1 in Figure 1 is a reactive (or deterministic) LTS. We only consider finite state, finite alphabet and finitely branching (i.e. finitely many transitions from any state) LPTSes.

We are also interested in LPTSes with a tree structure, i.e. the start state is not in the support of any distribution and every other state is in the support of exactly one distribution. We call such LPTSes *stochastic trees* or simply, *trees*.

We use $\langle S_i, s_i^0, \alpha_i, \tau_i \rangle$ for an LPTS L_i and $\langle S_L, s_L^0, \alpha_L, \tau_L \rangle$ for an LPTS L . The following notation is used in Section 5.

Notation 1 For an LPTS L and an alphabet α with $\alpha_L \subseteq \alpha$, L^α stands for the LPTS $\langle S_L, s_L^0, \alpha, \tau_L \rangle$.

Let L_1 and L_2 be two LPTSes and $\mu_1 \in \text{Dist}(S_1)$, $\mu_2 \in \text{Dist}(S_2)$.

Definition 2 (Product [20]). The product of μ_1 and μ_2 , denoted $\mu_1 \otimes \mu_2$, is a distribution in $\text{Dist}(S_1 \times S_2)$, such that $\mu_1 \otimes \mu_2 : (s_1, s_2) \mapsto \mu_1(s_1) \cdot \mu_2(s_2)$.

Definition 3 (Composition [20]). The parallel composition of L_1 and L_2 , denoted $L_1 \parallel L_2$, is defined as the LPTS $\langle S_1 \times S_2, (s_1^0, s_2^0), \alpha_1 \cup \alpha_2, \tau \rangle$ where $((s_1, s_2), a, \mu) \in \tau$ iff

1. $s_1 \xrightarrow{a} \mu_1$, $s_2 \xrightarrow{a} \mu_2$ and $\mu = \mu_1 \otimes \mu_2$, or

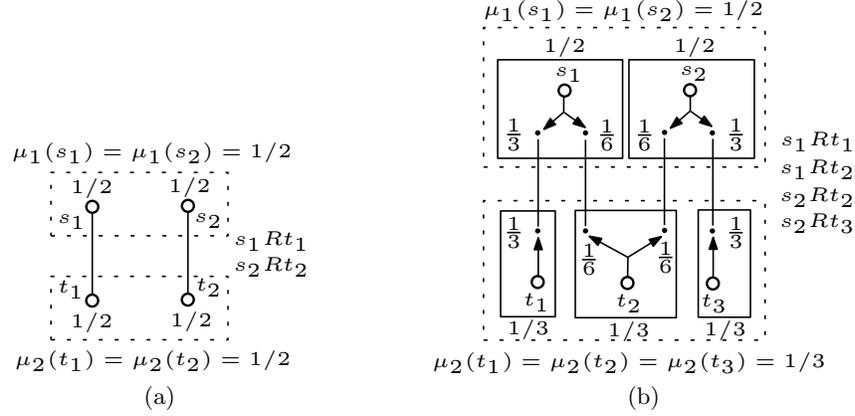


Fig. 2: Explaining $\mu_1 \sqsubseteq_R \mu_2$ by means of splitting (indicated by arrows) and matching (indicated by solid lines) the probabilities.

2. $s_1 \xrightarrow{a} \mu_1$, $a \notin \alpha_2$ and $\mu = \mu_1 \otimes \delta_{s_2}$, or
3. $a \notin \alpha_1$, $s_2 \xrightarrow{a} \mu_2$ and $\mu = \delta_{s_1} \otimes \mu_2$.

For example, in Figure 1, L is the composition of L_1 and L_2 .

Strong Simulation. For two LTSes, a pair of states belonging to a strong simulation relation depends on whether certain other pairs of successor states also belong to the relation [17]. For LPTSES, one has successor *distributions* instead of successor states; a pair of states belonging to a strong simulation relation R should now depend on whether certain other pairs in the *supports* of the successor distributions also belong to R . Therefore we define a binary relation on distributions, \sqsubseteq_R , which depends on the relation R between states. Intuitively, two distributions can be related if we can pair the states in their support sets, the pairs contained in R , *matching all* the probabilities under the distributions.

Consider an example with sRt and the transitions $s \xrightarrow{a} \mu_1$ and $t \xrightarrow{a} \mu_2$ with μ_1 and μ_2 as in Figure 2(a). In this case, one easy way to match the probabilities is to pair s_1 with t_1 and s_2 with t_2 . This is sufficient if $s_1 R t_1$ and $s_2 R t_2$ also hold, in which case, we say that $\mu_1 \sqsubseteq_R \mu_2$. However, such a direct matching may not be possible in general, as is the case in Figure 2(b). One can still obtain a matching by *splitting* the probabilities under the distributions in such a way that one can then directly match the probabilities as in Figure 2(a). Now, if $s_1 R t_1$, $s_1 R t_2$, $s_2 R t_2$ and $s_2 R t_3$ also hold, we say that $\mu_1 \sqsubseteq_R \mu_2$. Note that there can be more than one possible splitting. This is the central idea behind the following definition where the splitting is achieved by a *weight function*. Let $R \subseteq S_1 \times S_2$.

Definition 4 ([20]). $\mu_1 \sqsubseteq_R \mu_2$ iff there is a weight function $w : S_1 \times S_2 \rightarrow \mathbb{Q} \cap [0, 1]$ such that

1. $\mu_1(s_1) = \sum_{s_2 \in S_2} w(s_1, s_2)$ for all $s_1 \in S_1$,
2. $\mu_2(s_2) = \sum_{s_1 \in S_1} w(s_1, s_2)$ for all $s_2 \in S_2$,

3. $w(s_1, s_2) > 0$ implies $s_1 R s_2$ for all $s_1 \in S_1, s_2 \in S_2$.

$\mu_1 \sqsubseteq_R \mu_2$ can be checked by computing the maxflow in an appropriate network and checking if it equals 1.0 [1]. If $\mu_1 \sqsubseteq_R \mu_2$ holds, w in the above definition is one such maxflow function. As explained above, $\mu_1 \sqsubseteq_R \mu_2$ can be understood as *matching* all the probabilities (after splitting appropriately) under μ_1 and μ_2 . Considering $Supp(\mu_1)$ and $Supp(\mu_2)$ as two partite sets, this is the weighted analog of saturating a partite set in bipartite matching, giving us the following analog of the well-known Hall's Theorem for saturating $Supp(\mu_1)$.

Lemma 1 ([21]). $\mu_1 \sqsubseteq_R \mu_2$ iff for every $S \subseteq Supp(\mu_1)$, $\mu_1(S) \leq \mu_2(R(S))$.

It follows that when $\mu_1 \not\sqsubseteq_R \mu_2$, there exists a witness $S \subseteq Supp(\mu_1)$ such that $\mu_1(S) > \mu_2(R(S))$. For example, if $R(s_2) = \emptyset$ in Figure 2(a), its probability $\frac{1}{2}$ under μ_1 cannot be matched and $S = \{s_2\}$ is a witness subset.

Definition 5 (Strong Simulation [20]). R is a strong simulation iff for every $s_1 R s_2$ and $s_1 \xrightarrow{a} \mu_1^a$ there is a μ_2^a with $s_2 \xrightarrow{a} \mu_2^a$ and $\mu_1^a \sqsubseteq_R \mu_2^a$.

For $s_1 \in S_1$ and $s_2 \in S_2$, s_2 strongly simulates s_1 , denoted $s_1 \preceq s_2$, iff there is a strong simulation T such that $s_1 T s_2$. L_2 strongly simulates L_1 , also denoted $L_1 \preceq L_2$, iff $s_1^0 \preceq s_2^0$.

When checking a specification P of a system L with $\alpha_P \subset \alpha_L$, we implicitly assume that P is *completed* by adding Dirac self-loops on each of the actions in $\alpha_L \setminus \alpha_P$ from every state before checking $L \preceq P$. For example, $L \preceq P$ in Figure 1 assuming that P is completed with $\{send, ack\}$. Checking $L_1 \preceq L_2$ is decidable in polynomial time [1,21] and can be performed with a greatest fixed point algorithm that computes the coarsest simulation between L_1 and L_2 . The algorithm uses a relation variable R initialized to $S_1 \times S_2$ and checks the condition in Definition 5 for every pair in R , iteratively, removing any violating pairs from R . The algorithm terminates when a fixed point is reached showing $L_1 \preceq L_2$ or when the pair of initial states is removed showing $L_1 \not\preceq L_2$. If $n = \max(|S_1|, |S_2|)$ and $m = \max(|\tau_1|, |\tau_2|)$, the algorithm takes $O((mn^6 + m^2n^3)/\log n)$ time and $O(mn + n^2)$ space [1]. Several optimizations exist [21] but we do not consider them here, for simplicity.

We do consider a specialized algorithm for the case that L_1 is a tree which we use during abstraction refinement (Sections 4 and 5). It initializes R to $S_1 \times S_2$ and is based on a bottom-up traversal of L_1 . Let $s_1 \in S_1$ be a non-leaf state during such a traversal and let $s_1 \xrightarrow{a} \mu_1$. For every $s_2 \in S_2$, the algorithm checks if there exists $s_2 \xrightarrow{a} \mu_2$ with $\mu_1 \sqsubseteq_R \mu_2$ and removes (s_1, s_2) from R , otherwise, where R is the current relation. This constitutes an iteration in the algorithm. The algorithm terminates when (s_1^0, s_2^0) is removed from R or when the traversal ends. Correctness is not hard to show and we skip the proof.

Lemma 2 ([20]). \preceq is a preorder (i.e. reflexive and transitive) and is compositional, i.e. if $L_1 \preceq L_2$ and $\alpha_2 \subseteq \alpha_1$, then for every LPTS L , $L_1 \parallel L \preceq L_2 \parallel L$.

Finally, we show the *soundness* and *completeness* of the rule ASYM. The rule is *sound* if the conclusion holds whenever there is an A satisfying the premises. And the rule is *complete* if there is an A satisfying the premises whenever the conclusion holds.

Theorem 1. *For $\alpha_A \subseteq \alpha_2$, the rule ASYM is sound and complete.*

Proof. Soundness follows from Lemma 2. Completeness follows trivially by replacing A with L_2 . \square

3 Counterexamples to Strong Simulation

Let L_1 and L_2 be two LPTSes. We characterize a counterexample to $L_1 \preceq L_2$ as a tree and show that any simpler structure is not sufficient in general. We first describe counterexamples via a simple language-theoretic characterization.

Definition 6 (Language of an LPTS). *Given an LPTS L , we define its language, denoted $\mathcal{L}(L)$, as the set $\{L' \mid L' \text{ is an LPTS and } L' \preceq L\}$.*

Lemma 3. *$L_1 \preceq L_2$ iff $\mathcal{L}(L_1) \subseteq \mathcal{L}(L_2)$.*

Proof. Necessity follows trivially from the transitivity of \preceq and sufficiency follows from the reflexivity of \preceq which implies $L_1 \in \mathcal{L}(L_1)$. \square

Thus, a counterexample C can be defined as follows.

Definition 7 (Counterexample). *A counterexample to $L_1 \preceq L_2$ is an LPTS C such that $C \in \mathcal{L}(L_1) \setminus \mathcal{L}(L_2)$, i.e. $C \preceq L_1$ but $C \not\preceq L_2$.*

Now, L_1 itself is a trivial choice for C but it does not give any more useful information than what we had before checking the simulation. Moreover, it is preferable to have C with a special and simpler structure rather than a general LPTS as it helps in a more efficient counterexample analysis, wherever it is used. When the LPTSes are restricted to LTSes, a *tree-shaped* LTS is known to be sufficient as a counterexample [5]. Based on a similar intuition, we show that a stochastic tree is sufficient as a counterexample in the probabilistic case.

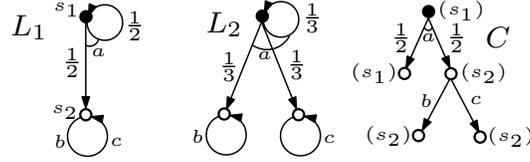
Theorem 2. *If $L_1 \not\preceq L_2$, there is a tree which serves as a counterexample.*

Proof. We only give a brief sketch of a constructive proof here. Counterexample generation is based on the coarsest strong simulation computation from Section 2. By induction on the size of the current relation R , we show that there is a tree counterexample to $s_1 \preceq s_2$ whenever (s_1, s_2) is removed from R . We only consider the inductive case here. The pair is removed because there is a transition $s_1 \xrightarrow{a} \mu_1$ but for every $s_2 \xrightarrow{a} \mu, \mu_1 \not\sqsubseteq_R \mu$ i.e. there exists $S_1^\mu \subseteq \text{Supp}(\mu_1)$ such that $\mu_1(S_1^\mu) > \mu(R(S_1^\mu))$. Such an S_1^μ can be found using Algorithm 1. Now, no pair in $S_1^\mu \times (\text{Supp}(\mu) \setminus R(S_1^\mu))$ is in R . By induction hypothesis, a counterexample tree exists for each such pair. A counterexample to $s_1 \preceq s_2$ is built using μ_1 and all these other trees. \square

Algorithm 1 Finding $T \subseteq S_1$ such that $\mu_1(T) > \mu(R(T))$.

 Given $\mu_1 \in \text{Dist}(S_1)$, $\mu \in \text{Dist}(S_2)$, $R \subseteq S_1 \times S_2$ with $\mu_1 \not\preceq_R \mu$.

- 1: let f be a maxflow function for the flow network corresponding to μ_1 and μ
 - 2: find $s_1 \in S_1$ with $\mu_1(s_1) > \sum_{s_2 \in S_2} f(s_1, s_2)$ and let $T = \{s_1\}$
 - 3: **while** $\mu_1(T) \leq \mu(R(T))$ **do**
 - 4: $T \leftarrow \{s_1 \in S_1 \mid \exists s_2 \in R(T) : f(s_1, s_2) > 0\}$
 - 5: **end while**
 - 6: **return** T
-


 Fig. 3: C is a counterexample to $L_1 \preceq L_2$.

For an illustration, see Figure 3 where C is a counterexample to $L_1 \preceq L_2$. Algorithm 1 is also analogous to the one used to find a subset failing Hall's condition in Graph Theory and can easily be proved correct. We obtain the following complexity bounds.

Theorem 3. *Deciding $L_1 \preceq L_2$ and obtaining a tree counterexample takes $O(mn^6 + m^2n^3)$ time and $O(mn + n^2)$ space where $n = \max(|S_{L_1}|, |S_{L_2}|)$ and $m = \max(|\tau_1|, |\tau_2|)$.*

Note that the obtained counterexample is essentially a finite *tree execution* of L_1 . That is, there is a total mapping $M : S_C \rightarrow S_1$ such that for every transition $c \xrightarrow{a} \mu_c$ of C , there exists $M(c) \xrightarrow{a} \mu_1$ such that M restricted to $\text{Supp}(\mu_c)$ is an injection and for every $c' \in \text{Supp}(\mu_c)$, $\mu_c(c') = \mu_1(M(c'))$. M is also a strong simulation. We call such a mapping an *execution mapping from C to L_1* . Figure 3 shows an execution mapping in brackets beside the states of C . We therefore have the following corollary.

Corollary 1. *If L_1 is reactive and $L_1 \not\preceq L_2$, there is a reactive tree which serves as a counterexample.*

The following two lemmas show that (reactive) trees are the simplest structured counterexamples.

Lemma 4. *There exist reactive LPTSes R_1 and R_2 such that $R_1 \not\preceq R_2$ and no counterexample is fully-probabilistic.*

Thus, if L_1 is reactive, a reactive tree is the simplest structure for a counterexample to $L_1 \preceq L_2$. This is surprising, since the non-probabilistic counterpart of a fully-probabilistic LPTS is a trace of actions and it is known that trace inclusion coincides with simulation conformance between reactive (*i.e.* deterministic) LPTSes. If there is no such restriction on L_1 , one may ask if a reactive LPTS suffices

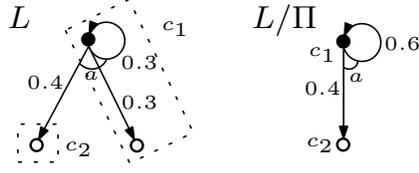


Fig. 4: An LPTS L , partition $\Pi = \{c_1, c_2\}$ and the quotient L/Π .

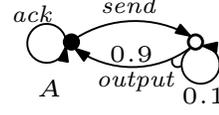


Fig. 5: An assumption for L_1 , L_2 and P in Figure 1.

as a counterexample to $L_1 \preceq L_2$. That is not the case either, as the following lemma shows.

Lemma 5. *There exist an LPTS L and a reactive LPTS R such that $L \not\preceq R$ and no counterexample is reactive.*

4 CEGAR for Checking Strong Simulation

Now that the notion of a counterexample has been formalized, we describe a CounterExample Guided Abstraction Refinement (CEGAR) approach [6] to check $L \preceq P$ where L and P are LPTSes and P stands for a *specification* of L . We will use this approach to describe AGAR in the next section.

Abstractions for L are obtained using a quotient construction from a *partition* Π of S_L . We let Π also denote the corresponding set of equivalence classes and given an arbitrary $s \in S$, let $[s]_\Pi$ denote the equivalence class containing s . The quotient is an adaptation of the usual construction in the non-probabilistic case.

Definition 8 (Quotient LPTS). *Given a partition Π of S_L , define the quotient LPTS, denoted L/Π , as the LPTS $\langle \Pi, [s_L^0]_\Pi, \alpha_L, \tau \rangle$ where $(c, a, \mu_l) \in \tau$ iff $(s, a, \mu) \in \tau_L$ for some $s \in S_L$ with $s \in c$ and $\mu_l(c') = \sum_{t \in c'} \mu(t)$ for all $c' \in \Pi$.*

As the abstractions are built from an explicit representation of L , this is not immediately useful. But, as we will see in Sections 5 and 6, this becomes very useful when adapted to the assume-guarantee setting.

Figure 4 shows an example quotient. Note that $L \preceq L/\Pi$ for any partition Π of S_L , with the relation $R = \{(s, c) | s \in c, c \in \Pi\}$ as a strong simulation.

CEGAR for LPTSes is sketched in Algorithm 2. It maintains an abstraction A of L , initialized to the quotient for the coarsest partition, and iteratively refines A based on the counterexamples obtained from the simulation check against P until a partition whose corresponding quotient conforms to P w.r.t. \preceq is obtained, or a real counterexample is found. In the following, we describe how to analyze if a counterexample is *spurious*, due to abstraction, and how to refine the abstraction in case it is (lines 4 – 6). Our analysis is an adaptation of an existing one for counterexamples which are arbitrary *sub-structures* of A [3]; while our tree counterexamples have an execution mapping to A , they are not necessarily sub-structures of A .

Analysis and Refinement (*analyzeAndRefine*(\cdot)). Assume that Π is a partition of S_L such that $A = L/\Pi$ and $A \not\preceq P$. Let C be a tree counterexample

Algorithm 2 CEGAR for LPTSeS: checks $L \preceq P$

```

1:  $A \leftarrow L/\Pi$ , where  $\Pi$  is the coarsest partition of  $S_L$ 
2: while  $A \not\preceq P$  do
3:   obtain a counterexample  $C$ 
4:    $(spurious, A') \leftarrow analyzeAndRefine(C, A, L)$  {see text}
5:   if spurious then
6:      $A \leftarrow A'$ 
7:   else
8:     return counterexample  $C$ 
9:   end if
10: end while
11: return  $L \preceq P$  holds

```

obtained by the algorithm described in Section 3, *i.e.* $C \preceq A$ but $C \not\preceq P$. As described in Section 3, there is an *execution mapping* $M : S_C \rightarrow S_A$ which is also a strong simulation. Let $R_M \subseteq S_C \times S_L$ be $\{(s_1, s_2) \mid s_1 M[s_2]_{\Pi}\}$. Our refinement strategy tries to obtain the coarsest strong simulation between C and L contained in R_M , using the specialized algorithm for trees described in Section 2 with R_M as the initial candidate. Let R and R_{old} be the candidate relations at the end of the current and the previous iterations, respectively, and let $s_1 \xrightarrow{a} \mu_1$ be the transition in C considered by the algorithm in the current iteration. (R_{old} is undefined initially.) The strategy refines a state when one of the following two cases happens before termination and otherwise, returns C as a *real* counterexample.

1. $R(s_1) = \emptyset$. There are two possible reasons for this case. One is that the states in $Supp(\mu_1)$ are not related, by R , to enough number of states in S_L (*i.e.* μ_1 is *spurious*) and (the images under M of) all the states in $Supp(\mu_1)$ are candidates for refinement. The other possible reason is the branching (more than one transition) from s_1 where no state in $R_M(s_1)$ can *simulate all* the transitions of s_1 and $M(s_1)$ is a candidate for refinement.
2. $M(s_1) = [s_L^0]_{\Pi}$, $s_L^0 \in R_{old}(s_1) \setminus R(s_1)$ and $R(s_1) \neq \emptyset$, *i.e.* $M(s_1)$ is the initial state of A but s_1 is no longer related to s_L^0 by R . Here, $M(s_1)$ is a candidate for refinement.

In case 1, our refinement strategy first tries to split the equivalence class $M(s_1)$ into $R_{old}(s_1)$ and the rest and then, for every state $s \in Supp(\mu_1)$, tries to split the equivalence class $M(s)$ into $R_{old}(s)$ and the rest, unless $M(s) = M(s_1)$ and $M(s_1)$ has already been split. And in case 2, the strategy splits the equivalence class $M(s_1)$ into $R_{old}(s_1) \setminus R(s_1)$ and the rest. It follows from the two cases that if C is declared real, then $C \preceq L$ with the final R as a strong simulation between C and L and hence, C is a counterexample to $L \preceq P$. The following lemma shows that the refinement strategy always leads to progress.

Lemma 6. *The above refinement strategy always results in a strictly finer partition $\Pi' < \Pi$.*

5 Assume-Guarantee Abstraction Refinement

We now describe our approach to Assume-Guarantee Abstraction Refinement (AGAR) for LPTSeS. The approach is similar to CEGAR from the previous section with the notable exception that counterexample analysis is performed in an assume guarantee style: a counterexample obtained from checking one component is used to refine the abstraction of a different component.

Given LPTSeS L_1 , L_2 and P , the goal is to check $L_1 \parallel L_2 \preceq P$ in an assume-guarantee style, using rule ASYM. The basic idea is to maintain A in the rule as an abstraction of L_2 , *i.e.* the second premise holds for free throughout, and to check only the first premise for every A generated by the algorithm. As in CEGAR, we restrict A to the quotient for a partition of S_2 . If the first premise holds for an A , then $L_1 \parallel L_2 \preceq P$ also holds, by the soundness of the rule. Otherwise, the obtained counterexample C is analyzed to see whether it indicates a real error or it is spurious, in which case A is refined (as described in detail below). Algorithm 3 sketches the AGAR loop.

For an example, A in Figure 5 shows the final assumption generated by AGAR for the LPTSeS in Figure 1 (after one refinement).

Algorithm 3 AGAR for LPTSeS: checks $L_1 \parallel L_2 \preceq P$

```

1:  $A \leftarrow$  coarsest abstraction of  $L_2$ 
2: while  $L_1 \parallel A \not\preceq P$  do
3:   obtain a counterexample  $C$ 
4:   obtain projections  $C \upharpoonright_{L_1}$  and  $C \upharpoonright_A$ 
5:    $(\text{spurious}, A') \leftarrow \text{analyzeAndRefine}(C \upharpoonright_A, A, L_2)$ 
6:   if spurious then
7:      $A \leftarrow A'$ 
8:   else
9:     return counterexample  $C$ 
10:  end if
11: end while
12: return  $L_1 \parallel L_2 \preceq P$  holds
    
```

Analysis and Refinement. The counterexample analysis is performed compositionally, using the *projections* of C onto L_1 and A . As there is an *execution mapping* from C to $L_1 \parallel A$, these projections are the *contributions* of L_1 and A towards C in the composition. We denote these projections by $C \upharpoonright_{L_1}$ and $C \upharpoonright_A$, respectively. In the non-probabilistic case, these are obtained by simply projecting C onto the respective alphabets. In the probabilistic scenario, however, composition changes the probabilities in the distributions (Definition 2) and alphabet projection is insufficient. For this reason, we additionally record the individual distributions of the LPTSeS responsible for a product distribution while performing the composition. Thus, projections $C \upharpoonright_{L_1}$ and $C \upharpoonright_A$ can be obtained using this auxiliary information. Note that there is a natural *execution mapping* from $C \upharpoonright_A$ to A and from $C \upharpoonright_{L_1}$ to L_1 . We can then employ the analysis described in Section 4 between $C \upharpoonright_A$ and A , *i.e.* invoke $\text{analyzeAndRefine}(C \upharpoonright_A, A, L_2)$ to determine if $C \upharpoonright_A$ (and hence, C) is spurious and to refine A in case it is. Otherwise,

$C \upharpoonright_A \preceq L_2$ and hence, $(C \upharpoonright_A)^{\alpha_2} \preceq L_2$. Together with $(C \upharpoonright_{L_1})^{\alpha_1} \preceq L_1$ this implies $(C \upharpoonright_{L_1})^{\alpha_1} \parallel (C \upharpoonright_A)^{\alpha_2} \preceq L_1 \parallel L_2$ (Lemma 2). As $C \preceq (C \upharpoonright_{L_1})^{\alpha_1} \parallel (C \upharpoonright_A)^{\alpha_2}$, C is then a *real* counterexample. Thus, we have the following result.

Theorem 4 (Correctness and Termination). *Algorithm AGAR always terminates with at most $|S_2| - 1$ refinements and $L_1 \parallel L_2 \not\preceq P$ if and only if the algorithm returns a real counterexample.*

Proof. Correctness: AGAR terminates when either Premise 1 is satisfied by the current assumption (line 12) or when a counterexample is returned (line 9). In the first case, we know that Premise 2 holds by construction and since ASYM is sound (Theorem 1) it follows that indeed $L_1 \parallel L_2 \preceq P$. In the second case, the counterexample returned by AGAR is real (see above) showing that $L_1 \parallel L_2 \not\preceq P$.

Termination: AGAR iteratively refines the abstraction until the property holds or a real counterexample is reported. Abstraction refinement results in a finer partition (Lemma 6) and thus it is guaranteed to terminate since in the worst case A converges to L_2 which is finite state. Since rule ASYM is trivially complete for L_2 (proof of Theorem 1) it follows that AGAR will also terminate, and the number of refinements is bounded by $|S_2| - 1$. \square

In practice, we expect AGAR to terminate earlier than in $|S_2| - 1$ steps, with an assumption smaller than L_2 . AGAR will terminate as soon as it finds an assumption that satisfies the premises or that helps exhibit a real counterexample. Note also that, although AGAR uses an explicit representation for the individual components, it never builds $L_1 \parallel L_2$ directly (except in the worst-case) keeping the cost of verification low.

Reasoning with $n \geq 2$ Components. So far, we have discussed compositional verification in the context of two components L_1 and L_2 . This reasoning can be generalized to $n \geq 2$ components using the following (sound and complete) rule.

$$\frac{1 : L_1 \parallel A_1 \preceq P \quad 2 : L_2 \parallel A_2 \preceq A_1 \quad \dots \quad n : L_n \preceq A_{n-1}}{\parallel_{i=1}^n L_i \preceq P} \text{ (ASYM-N)}$$

The rule enables us to overcome the *intermediate state explosion* that may be associated with two-way decompositions (when the subsystems are larger than the entire system). The AGAR algorithm for this rule involves the creation of $n - 1$ nested instances of AGAR for two components, with the i th instance computing the assumption A_i for $(L_1 \parallel \dots \parallel L_i) \parallel (L_{i+1} \parallel A_{i+1}) \preceq P$. When the AGAR instance for A_{i-1} returns a counterexample C , for $1 < i \leq n - 1$, we need to analyze C for spuriousness and refine A_i in case it is. From Algorithm 3, C is returned only if *analyzeAndRefine* $(C \upharpoonright_{A_{i-1}}, A_{i-1}, L_i \parallel A_i)$ concludes that $C \upharpoonright_{A_{i-1}}$ is real (note that A_{i-1} is an abstraction of $L_i \parallel A_i$). From *analyzeAndRefine* in Section 4, this implies that the final relation R computed between the states of $C \upharpoonright_{A_{i-1}}$ and $L_i \parallel A_i$ is a strong simulation between them. It follows that, although $C \upharpoonright_{A_{i-1}}$ does not have an *execution mapping* to $L_i \parallel A_i$, we can naturally obtain a tree T using $C \upharpoonright_{A_{i-1}}$, via R , with such a mapping. Thus, we modify the algorithm to return $T \upharpoonright_{A_i}$ at line 9, instead of C , which can then be

used to check for spuriousness and refine A_i . Note that when A_i is refined, all the A_j 's for $j < i$ need to be recomputed.

Compositional Verification of Logical Properties. AGAR can be further applied to automate assume-guarantee checking of properties ϕ written as formulae in a logic that is preserved by strong simulation such as the *weak-safety* fragment of probabilistic CTL (pCTL) [3] which also yield trees as counterexamples. The rule ASYM is both sound and complete for this logic (\models denotes property satisfaction) for $\alpha_A \subseteq \alpha_2$ with a proof similar to that of Theorem 1.

$$\frac{1 : L_1 \parallel A \models \phi \quad 2 : L_2 \preceq A}{L_1 \parallel L_2 \models \phi}$$

A can be computed as a conservative abstraction of L_2 and iteratively refined based on the tree counterexamples to premise 1, using the same procedures as before. The rule can be generalized to reasoning about $n \geq 2$ components as described above and also to richer logics with more general counterexamples adapting existing CEGAR approaches [3] to AGAR. We plan to further investigate this direction in the future.

6 Implementation and Results

Implementation. We implemented the algorithms for checking simulation (Section 2), for generating counterexamples (as in the proof of Lemma 2) and for AGAR (Algorithm 3) with ASYM and ASYM-N in JavaTM. We used the front-end of PRISM's [15] explicit-state engine to parse the models of the components described in PRISM's input language and construct LPTSeS which were then handled by our implementation.

While the JavaTM implementation for checking simulation uses the greatest fixed point computation to obtain the coarsest strong simulation, we noticed that the problem of checking the existence of a strong simulation is essentially a constraint satisfaction problem. To leverage the efficient constraint solvers that exist today, we reduced the problem of checking simulation to an SMT problem with rational linear arithmetic as follows. For every pair of states, the constraint that the pair is in some strong simulation is simply the encoding of the condition in Definition 5. For a relevant pair of distributions μ_1 and μ_2 , the constraint for $\mu_1 \sqsubseteq_R \mu_2$ is encoded by means of a weight function (as given by Definition 4) and the constraint for $\mu_1 \not\sqsubseteq_R \mu_2$ is encoded by means of a witness subset of $Supp(\mu_1)$ (as in Lemma 1), where R is the variable for the strong simulation. We use Yices (v1.0.29) [9] to solve the resulting SMT problem; a *real* variable in Yices input language is essentially a rational variable. There is no direct way to obtain a tree counterexample when the SMT problem is unsatisfiable. Therefore when the conformance fails, we obtain the *unsat core* from Yices, construct the *sub-structure* of L_1 (when we check $L_1 \preceq L_2$) from the constraints in the unsat core and check the conformance of this sub-structure against L_2 using the JavaTM implementation. This sub-structure is usually much smaller than L_1 and contains only the information necessary to expose the counterexample.

Example (param)	L	P	ASYM						ASYM-N						MONO	
			L ₁	L ₂	Time	Mem	L _M	A _M	L _c	Time	Mem	L _M	A _M	Time	Mem	
CS ₁ (5)	94	16	36	405	7.2	15.6	182	33	36	74.0	15.1	182	34	0.2	8.8	
CS ₁ (6)	136	19	49	1215	11.6	22.7	324	41	49	810.7	21.4	324	40	0.5	12.2	
CS ₁ (7)	186	22	64	3645	37.7	49.4	538	56	64	out	–	–	–	0.8	17.9	
CS _N (2)	34	15	25	9	0.7	7.1	51	7	9	2.4	6.8	40	25	0.1	5.9	
CS _N (3)	184	54	125	16	43.0	63.0	324	12	16	1.6k	109.6	372	125	14.8	37.9	
CS _N (4)	960	189	625	25	out	–	–	–	25	out	–	–	–	1.8k	667.5	
MER(3)	16k	12	278	1728	2.6	19.7	706	7	278	3.6	14.6	706	7	193.8	458.5	
MER(4)	120k	15	465	21k	15.0	53.9	2k	11	465	34.7	37.8	2k	11	out	–	
MER(5)	841k	18	700	250k	–	out ¹	–	–	700	257.8	65.5	3.3k	16	–	out ¹	
SN(1)	462	18	43	32	0.2	6.2	43	3	126	1.7	8.5	165	6	1.5	27.7	
SN(2)	7860	54	796	32	79.5	112.9	796	3	252	694.4	171.7	1.4k	21	4.7k	1.3k	
SN(3)	78k	162	7545	32	out	–	–	–	378	7.2k	528.8	1.4k	21	–	out	

Table 1: AGAR vs monolithic verification. ¹ Mem-out during model construction.

Results. We evaluated our algorithms using this implementation on several examples analyzed in previous work [11]. Some of these examples were created by introducing probabilistic failures into non-probabilistic models used earlier [19] while others were adapted from PRISM benchmarks [15]. The properties used previously were about *probabilistic reachability* and we had to create our own specification LPTSes after developing an understanding of the models. The models in all the examples satisfy the respective specifications. We briefly describe the models and the specifications below, all of which are available at <http://www.cs.cmu.edu/~akomurav/publications/agar/AGAR.html>.

CS₁ and CS_N model a *Client-Server* protocol with mutual exclusion having probabilistic failures in one or all of the N clients, respectively. The specifications describe the probabilistic failure behavior of the clients while hiding some of the actions as is typical in a high level design specification.

MER models a *resource arbiter* module of NASA’s software for *Mars Exploration Rovers* which grants and rescinds shared resources for several users. We considered the case of two resources with varying number of users and probabilistic failures introduced in all the components. As in the above example, the specifications describe the probabilistic failure behavior of the users while hiding some of the actions.

SN models a wireless *Sensor Network* of one or more sensors sending data and messages to a process via a channel with a bounded buffer having probabilistic behavior in the components. Creating specification LPTSes for this example turned out to be more difficult than the above examples, and we obtained them by observing the system’s runs and by manual abstraction.

Table 1 shows the results we obtained when ASYM and ASYM-N were compared with monolithic (non-compositional) conformance checking. $|X|$ stands for the number of states of an LPTS X . L stands for the whole system, P for the specification, L_M for the LPTS with the largest number of states built by composing LPTSes during the course of AGAR, A_M for the assumption with the largest number of states during the execution and L_c for the component with the largest number of states in ASYM-N. *Time* is in seconds and *Memory* is in megabytes. We also compared $|L_M|$ with $|L|$, as $|L_M|$ denotes the largest LPTS ever built by AGAR. Best figures, among ASYM, ASYM-N and MONO,

for *Time*, *Memory* and LPTS sizes, are boldfaced. All the results were taken on a Fedora-10 64-bit machine running on an Intel® Core™2 Quad CPU of 2.83GHz and 4GB RAM. We imposed a 2GB upper bound on Java heap memory and a 2 hour upper bound on the running time. We observed that most of the time during AGAR was spent in checking the premises and an insignificant amount was spent for the composition and the refinement steps. Also, most of the memory was consumed by Yices. We tried several orderings of the components (the L_i 's in the rules) and report only the ones giving the best results.

While monolithic checking outperformed AGAR for *Client-Server*, there are significant time and memory savings for *MER* and *Sensor Network* where in some cases the monolithic approach ran out of resources (time or memory). One possible reason for AGAR performing worse for *Client-Server* is that $|L|$ is much smaller than $|L_1|$ or $|L_2|$. When compared to using ASYM, ASYM-N brings further memory savings in the case of *MER* and also time savings for *Sensor Network* with parameter 3 which could not finish in 2 hours when used with ASYM. As already mentioned, these models were analyzed previously with an assume-guarantee framework using learning from traces [11]. Although that approach uses a similar assume-guarantee rule (but instantiated to check *probabilistic reachability*) and the results have some similarity (e.g. *Client-Server* is similarly not handled well by the compositional approach), we can not directly compare it with AGAR as it considers a different class of properties.

7 Conclusion and Future Work

We described a complete, fully automated abstraction-refinement approach for assume-guarantee checking of strong simulation between LPTSes. The approach uses refinement based on counterexamples formalized as stochastic trees and it further applies to checking *safe*-pCTL properties. We showed experimentally the merits of the proposed technique. We plan to extend our approach to cases where the assumption A has a smaller alphabet than that of the component it represents as this can potentially lead to further savings. Strong simulation would no longer work and one would need to use *weak* simulation [20], for which checking algorithms are unknown yet. We would also like to explore symbolic implementations of our algorithms, for increased scalability. As an alternative approach, we plan to build upon our recent work [14] on learning LPTSes to develop practical compositional algorithms and compare with AGAR.

Acknowledgments. We thank Christel Baier, Rohit Chadha, Lu Feng, Holger Hermanns, Marta Kwiatkowska, Joel Ouaknine, David Parker, Frits Vaandrager, Mahesh Viswanathan, James Worrell and Lijun Zhang for generously answering our questions related to this research. We also thank the anonymous reviewers for their suggestions and David Henriques for carefully reading an earlier draft.

References

1. C. Baier. On Algorithmic Verification Methods for Probabilistic Systems. Habilitation thesis, Fakultät für Mathematik und Informatik, Univ. Mannheim, 1998.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, MA, USA, 2008.
3. R. Chadha and M. Viswanathan. A Counterexample-Guided Abstraction-Refinement Framework for Markov Decision Processes. *TOCL*, 12(1):1–49, 2010.
4. S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. In *CAV*, volume 3576 of *LNCS*, 2005.
5. S. J. Chaki. *A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs*. PhD thesis, Carnegie Mellon University, 2005.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, volume 1855 of *LNCS*, 2000.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 2000.
8. L. de Alfaro, T. A. Henzinger, and R. Jhala. Compositional Methods for Probabilistic Systems. In *CONCUR*, volume 2154 of *LNCS*, 2001.
9. B. Dutertre and L. D. Moura. The Yices SMT Solver. Technical report, SRI International, 2006.
10. L. Feng, T. Han, M. Kwiatkowska, and D. Parker. Learning-based Compositional Verification for Synchronous Probabilistic Systems. In *ATVA*, volume 6996 of *LNCS*, 2011.
11. L. Feng, M. Kwiatkowska, and D. Parker. Automated learning of probabilistic assumptions for compositional reasoning. In *FASE*, volume 6603 of *LNCS*, 2011.
12. M. Gheorghiu Bobaru, C. S. Păsăreanu, and D. Giannakopoulou. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *CAV*, volume 5123 of *LNCS*, 2008.
13. H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In *CAV*, volume 5123 of *LNCS*, 2008.
14. A. Komuravelli, C. S. Păsăreanu, and E. M. Clarke. Learning Probabilistic Systems from Tree Samples. In *LICS*, 2012. (to appear).
15. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV*, volume 6806 of *LNCS*, 2011.
16. M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-Guarantee Verification for Probabilistic Systems. In *TACAS*, volume 6015 of *LNCS*, 2010.
17. R. Milner. An Algebraic Definition of Simulation between Programs. Technical report, Stanford University, 1971.
18. A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *LMCS*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, 1985.
19. C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning. *FMSD*, 32(3):175–205, 2008.
20. R. Segala and N. Lynch. Probabilistic Simulations for Probabilistic Processes. *Nordic J. of Computing*, 2(2):250–273, 1995.
21. L. Zhang. *Decision Algorithms for Probabilistic Simulations*. PhD thesis, Universität des Saarlandes, 2008.