Proving Termination of Probabilistic Programs Using Patterns

Javier Esparza[1], Andreas Gaiser[1,⋆], and Stefan Kiefer[2,⋆⋆]

[1] Institut für Informatik, Technische Universität München, Germany
{esparza,gaiser}@model.in.tum.de
[2] Department of Computer Science, University of Oxford, United Kingdom
stefan.kiefer@cs.ox.ac.uk

Abstract. Proving programs terminating is a fundamental computer science challenge. Recent research has produced powerful tools that can check a wide range of programs for termination. The analog for probabilistic programs, namely termination with probability one ("almost-sure termination"), is an equally important property for randomized algorithms and probabilistic protocols. We suggest a novel algorithm for proving almost-sure termination of probabilistic programs. Our algorithm exploits the power of state-of-the-art model checkers and termination provers for nonprobabilistic programs: it calls such tools within a refinement loop and thereby iteratively constructs a "terminating pattern", which is a set of terminating runs with probability one. We report on various case studies illustrating the effectiveness of our algorithm. As a further application, our algorithm can improve lower bounds on reachability probabilities.

1 Introduction

Proving program termination is a fundamental challenge of computer science. Termination is expressible in temporal logic, and so checkable in principle by LTL or CTL model-checkers. However, recent research has shown that special purpose tools, like Terminator and ARMC [18,4], and techniques like *transition invariants*, can be dramatically more efficient [17,20,19].

The analog of termination for probabilistic programs is termination with probability one, or *almost sure termination*, abbreviated here to *a.s.-termination*. Since a.s.-termination is as important for randomized algorithms and probabilistic protocols as termination is for regular programs, the question arises whether the very strong advances in automatic termination proving termination can be exploited in the probabilistic case. However, it is not difficult to see that, without further restricting the question, the answer is negative. The reason is that termination is a purely topological property of the transition system associated

⋆ Andreas Gaiser is supported by the DFG Graduiertenkolleg 1480 (PUMA).
⋆⋆ Stefan Kiefer is supported by a postdoctoral fellowship of the German Academic Exchange Service (DAAD).

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 123–138, 2012.

to the program, namely absence of cycles, but a.s.-termination is not. Consider for instance the program

```
k = 1;  while (0 < k) { if coin(p) k++ else k-- }
```

where `coin(p)` yields 1 with probability $0 < p < 1$, and 0 with probability $(1 - p)$. The program has the same executions for all values of p (only their probabilities change), but it only terminates a.s. for $p \leq 1/2$. This shows that proving a.s.-termination requires arithmetic reasoning not offered by termination provers.

The situation changes if we restrict our attention to *weakly finite* probabilistic programs. Loosely speaking, a program is weakly finite if the set of states reachable from any initial state is finite. Notice that the state space may be infinite, because the set of initial states may be infinite. Weakly finite programs are a large class, which in particular contains *parameterized probabilistic programs*, i.e., programs with parameters that can be initialized to arbitrary large values, but are finite-state for every valuation of the parameters. One can show that a.s.-termination is a topological property for weakly finite programs. If the program does not contain nondeterministic choices, then it terminates a.s. iff for every reachable state s there is a path leading from s to a terminating state, which corresponds to the CTL property $AG\,EF\,end$. (In the nondeterministic case there is also a corresponding topological property.) As in the nonprobabilistic case, generic infinite-state model checkers perform poorly for these properties because of the quantifier alternation $AG\,EF$. In particular, CEGAR approaches usually fail, because, crudely speaking, they tend to unroll loops, which is essentially useless for proving termination.

In [1], Arons, Pnueli and Zuck present a different and very elegant approach that reduces *a.s.-termination* of a *probabilistic* program to *termination* of a *nondeterministic* program obtained with the help of a *Planner*. A Planner occasionally and infinitely often determines the outcome of the next k random choices for some fixed k, while the other random choices are performed nondeterministically. The planner approach is based on the following simple proof rule, with P a probabilistic program and R a measurable set of runs of P:

$$\frac{Pr[R] = 1 \qquad \text{Every } r \in R \text{ is terminating}}{P \text{ terminates a.s.}}$$

In this paper we revisit and generalize this approach, with the goal of profiting from recent advances on termination tools and techniques not available when [1] was published. While we also partially fix the outcome of random choices, we do so more flexibly with the help of *patterns*. A first advantage of patterns is that we are able to obtain a *completeness* result for weakly finite programs, which is not the case for Planners. Further, in contrast to [1], we show how to automatically derive patterns for finite-state and weakly finite programs using an adapted version of the CEGAR approach. Finally, we apply our technique to improve CEGAR-algorithms for *quantitative* probabilistic verification [7,8,10,5].

In the rest of this introduction we explain our approach by means of examples. First we discuss finite-state programs and then the weakly finite case.

Finite-state programs. Consider the finite-state program FW shown on the left of Fig. 1. It is an abstraction of part of the FireWire protocol [12]. Loosely speaking,

```
k = 0;
while (k < 100) {
  old_x = x;
  x = coin(p);
  if (x != old_x) k++
}
```

```
c1 = ?; c2 = 2;
k = 0;
while (k < 100) {
  old_x = x;
  if (c1 > 0)      { x = nondet(); c1-- }
  elseif (c2 = 2 ) { x = 0; c2-- }
  elseif (c2 = 1 ) { x = 1; c2-- }
  else /* c1 = 0 and c2 = 0 */ { c1 = ?; c2 = 2 }
  if (x != old_x) k++
}
```

Fig. 1. The programs FW and FW'

FW terminates a.s. because if we keep tossing a coin then with probability 1 we observe 100 times two consecutive tosses with the opposite outcome (we even see 100 times the outcome 01). More formally, let $C = \{0, 1\}$, and let us identify a *run* of FW (i.e., a terminating or infinite execution) with the sequence of 0's and 1's corresponding to the results of the coin tosses carried out during it. For instance, $(01)^{51}$ and $(001100)^{50}$ are terminating runs of FW, and 0^ω is a nonterminating run. FW terminates because the runs that are prefixes of $(C^*01)^\omega$ have probability 1, and all of them terminate. But it is easy to see that these are also the runs of the nondeterministic program FW' on the right of Fig. 1 where `c = ?` nondeterministically sets `c` to an arbitrary nonnegative integer. Since termination of FW' can easily be proved with the help of ARMC, we have proved a.s.-termination of FW.

We present an automatic procedure leading from FW to FW' based on the notion of *patterns*. A pattern is a subset of C^ω of the form $C^*w_1C^*w_2C^*w_3\ldots$, where $w_1, w_2, \ldots \in C^*$. We call a pattern *simple* if it is of the form $(C^*w)^\omega$. A pattern Φ is *terminating (for a probabilistic program P)* if all runs of P that *conform* to Φ, i.e., that are prefixes of words of Φ, terminate. In the paper we prove the following theorems:

(1) For every pattern Φ and program P, the Φ-conforming runs of P have probability 1.
(2) Every finite-state program has a simple terminating pattern.

By these results, we can show that FW terminates a.s. by finding a simple terminating pattern Φ, taking for P' a nondeterministic program whose runs are the Φ-conforming runs of P, and proving that P' terminates. In the paper we show how to automatically find Φ with the help of a finite-state model-checker (in our experiments we use SPIN). We sketch the procedure using FW as example. First we check if some run of FW conforms to $\Phi_0 = C^\omega$, i.e., if some run of FW is infinite, and get $v_1 = 0^\omega$ as answer. Using an algorithm provided in the paper, we compute a *spoiler* w_1 of v_1: a finite word that is not an infix of v_1. The algorithm

yields $w_1 = 1$. We now check if some run of `FW` conforms to $\Phi_1 = (C^* w_1)^\omega$, and get $v_2 = 1^\omega$ as counterexample, and construct a spoiler w_2 of *both* v_1 and v_2: a finite word that is an infix of *neither* v_1^ω *nor* v_2^ω. We get $w_2 = 01$, and check if some run of `FW` conforms to $\Phi_2 = (C^* w_2)^\omega$. The checker finds no counterexamples, and so Φ_2 is terminating. In the paper we prove that the procedure is complete, i.e., produces a terminating pattern for any finite-state program that terminates a.s.

Weakly finite programs. We now address the main goal of the paper: proving a.s.-termination for weakly finite programs. Unfortunately, Proposition (2) no longer holds. Consider the random-walk program `RW` on the left of Fig. 2, where N is an input variable. `RW` terminates a.s., but we can easily show (by setting N

```
k = 1;
while (0 < k < N) {
  if coin(p) k++ else k--
}
```

```
K = 2; c1 = ?; c2 = K;
k = 1;
while (0 < k < N) {
    if (c1 > 0) {
      if nondet() k++ else k--; c1--
    }
    elseif (c2 > 0) { k--; c2-- }
    else { K++; c1 = ?; c2 = K }
}
```

Fig. 2. The programs `RW` and `RW'`

to a large enough value) that no simple pattern is terminating. However, there *is* a terminating pattern, namely $\Phi = C^*00C^*000C^*0000\ldots$: every Φ-conforming run terminates, whatever value N is set to. Since, by result (1), the Φ-conforming runs have probability 1 (intuitively, when tossing a coin we will eventually see longer and longer chains of 0's), `RW` terminates a.s. In the paper we show that this is not a coincidence by proving the following completeness result:

(3) Every weakly finite program has a (not necessarily simple) terminating pattern.

In fact, we even prove the existence of a *universal* terminating pattern, i.e., a single pattern Φ_u such that for all weakly finite, a.s.-terminating probabilistic programs all Φ_u-conforming runs terminate. This gives a universal reduction of a.s.-termination to termination, but one that is not very useful in practice. In particular, since the universal pattern *is* universal, it is not tailored towards making the proof of any particular program simple. For this reason we propose a technique that reuses the procedure for finite-state programs, and extends it with an extrapolation step in order to produce a candidate for a terminating pattern. We sketch the procedure using `RW` as example. Let $\texttt{RW}_i$ be the program `RW` with $N = i$. Since every $\texttt{RW}_i$ is finite-state, we can find terminating patterns $\Phi_i = (C^* u_i)^\omega$ for a finite set of values of i, say for $i = 1, 2, 3, 4, 5$. We obtain $u_1 = u_2 = \epsilon$, $u_3 = 00$, $u_4 = 000$, $u_5 = 000$. We prove in the paper that Φ_i is

not only terminating for $\texttt{RW}_i$, but also for every $\texttt{RW}_j$ with $j \leq i$. This suggests to extrapolate and take the pattern $\Phi = C^*00C^*000C^*0000\ldots$ as a candidate for a terminating pattern for RW. We automatically construct the nondeterministic program RW' on the right of Fig. 2. Again, ARMC proves that RW' terminates, and so that RW terminates a.s.

Related work. A.s.-termination is highly desirable for protocols if termination within a fixed number of steps is not feasible. For instance, [3] considers the problem of reaching consensus within a set of interconnected processes, some of which may be faulty or even malicious. They succeed in designing a probabilistic protocol to reach consensus a.s., although it is known that no deterministic algorithm terminates within a bounded number of steps. A well-known approach for proving a.s.-termination are Pnueli et al.'s notions of extreme fairness and α-fairness [15,16]. These proof methods, although complete for finite-state systems, are hard to automatize and require a lot of knowledge about the considered program. The same applies for the approach of McIver et al. in [11] that offers proof rules for probabilistic loops in pGCL, an extension of Dijkstra's guarded language. The paper [13] discusses probabilistic termination in an abstraction-interpretation framework. It focuses on programs with a (single) loop and proposes a method of proving that the probability of taking the loop k times decreases exponentially with k. This implies a.s.-termination. In contrast to our work there is no tool support in [13].

Organization of the paper. Sections 2 contains preliminaries and the syntax and semantics of our model of probabilistic programs. Section 3 proves soundness and completeness results for termination of weakly finite programs. Section 4 describes the iterative algorithm for generating patterns. Section 5 discusses case studies. Section 6 concludes. For space reasons, a full discussion of nondeterministic programs and some proofs are omitted. They can be found in the full version of the paper in [6].

2 Preliminaries

For a finite nonempty set Σ, we denote by Σ^* and Σ^ω the sets of finite and infinite words over Σ, and set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

Markov Decision Processes and Markov Chains. A *Markov Decision Process* (MDP) is a tuple $\mathcal{M} = (Q_A, Q_P, \mathrm{Init}, \to, \mathrm{Lab}_A, \mathrm{Lab}_P)$, where Q_A and Q_P are countable or finite sets of *action nodes* and *probabilistic nodes*, $\mathrm{Init} \subseteq Q_A \cup Q_P$ is a set of *initial nodes*, and Lab_A and Lab_P are disjoint, finite sets of *action labels* and *probabilistic labels*. Finally, the relation $\to$ is equal to $\to_A \cup \to_P$, where $\to_A \subseteq Q_A \times \mathrm{Lab}_A \times (Q_A \cup Q_P)$ is a set of *action transitions*, and $\to_P \subseteq Q_P \times (0,1] \times \mathrm{Lab}_P \times Q$ is a set of *probabilistic transitions* satisfying the following conditions: (a) if (q,p,l,q') and (q,p',l,q') are probabilistic transitions, then $p = p'$; (b) the probabilities of the outgoing transitions of a probabilistic node add up to 1. We also require that every node of Q_A has at least one successor in $\to_A$. If $Q_A = \emptyset$ and $\mathrm{Init} = \{q_I\}$ then we call $\mathcal{M}$ a *Markov chain* and

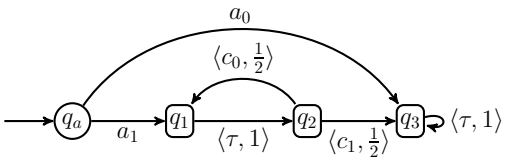

Fig. 3. Example MDP

write $\mathcal{M} = (Q_P, q_I, \to, \mathrm{Lab}_P)$. We set $Q = Q_A \cup Q_P$ and $\mathrm{Lab} = \mathrm{Lab}_A \cup \mathrm{Lab}_P$. We write $q \xrightarrow{l} q'$ for $(q, l, q') \in \to_A$, and $q \xrightarrow{l,p} q'$ for $(q, p, l, q') \in \to_P$ (we skip p if it is irrelevant). For $w = l_1 l_2 \dots l_n \in \mathrm{Lab}^*$, we write $q \xrightarrow{w} q'$ if there exists a path $q = q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_n} q_n = q'$.

Example 1. Figure 3 shows an example of a Markov Decision Process $\mathcal{M} = (\{q_a\}, \{q_1, q_2, q_3\}, \mathrm{Init}, \to, \mathrm{Lab}_A, \mathrm{Lab}_P)$, with action labels a_0, a_1, probabilistic labels τ, c_0, c_1, and a single initial node q_a.

Runs, Paths, Probability Measures, Traces. A *run* of an MDP $\mathcal{M}$ is an infinite word $r = q_0 l_0 q_1 l_1 \dots \in (Q\mathrm{Lab})^\omega$ such that for all $i \geq 0$ either $q_i \xrightarrow{l_i,p} q_{i+1}$ for some $p \in (0, 1]$ or $q_i \xrightarrow{l_i} q_{i+1}$. We call the run *initial* if $q_0 \in \mathrm{Init}$. We denote the set of runs starting at a node q by $\mathrm{Runs}^{\mathcal{M}}(q)$, and the set of all runs starting at initial nodes by $\mathrm{Runs}(\mathcal{M})$.

A *path* is a proper prefix of a run. We denote by $\mathrm{Paths}^{\mathcal{M}}(q)$ the set of all paths starting at q. We often write $r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \dots$ instead of $r = q_0 l_0 q_1 \dots$ for both runs and paths, and skip the superscripts of $\mathrm{Runs}(\cdot)$ and $\mathrm{Paths}(\cdot)$ if the context is clear.

We take the usual, cylinder-based definition of a probability measure $\Pr_{q_0}$ on the set of runs of a Markov chain $\mathcal{M}$ starting at a state $q_0 \in \mathrm{Init}$ (see e.g. [2] or [6]) for details). For general MDPs, we define a probability measure $\Pr^S_{q_0}$ with respect to a *strategy* S. We may drop the subscript if the initial state is irrelevant or understood.

The *trace* of a run $r = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \dots \in \mathrm{Runs}(\mathcal{M})$, denoted by $\bar{r}$, is the infinite sequence $\alpha_0 \alpha_1 \dots \in \mathrm{Lab}$ of labels. Given $\Sigma \subseteq \mathrm{Lab}$, we define $\bar{r}|_\Sigma$ as the projection of $\bar{r}$ onto Σ. Observe that $\bar{r}|_\Sigma$ can be finite.

2.1 Probabilistic Programs

We model probabilistic programs as flowgraphs whose transitions are labeled with *commands*. Since our model is standard and very similar to [10], we give an informal but hopefully precise enough definition. Let Var be a set of variable names over the integers (the variable domain could be easily extended), and let *Val* be the set of possible *valuations* of Var, also called *configurations*. The set of commands contains

– conditional statements, i.e., boolean combinations of expressions $e \leq e'$, where e, e' are arithmetic expressions (e.g, $x + y \leq 5 \wedge y \geq 3$);

- deterministic assignments $x := e$ and nondeterministic assignments $x :=$ nondet() that nondeterministically assign to x the value 0 or 1;
- probabilistic assignments $x :=$ coin(p) that assign to x the value 0 or 1 with probability p or $(1-p)$, respectively.

A *probabilistic program* P is a tuple $(\mathcal{L}, I, \hookrightarrow, \text{label}, \bot, \top)$, where $\mathcal{L}$ is a finite set of control flow *locations*, $I \subseteq \mathit{Val}$ is a set of *initial configurations*, $\hookrightarrow \subseteq \mathcal{L} \times \mathcal{L}$ is the *flow relation* (as usual we write $l \hookrightarrow l'$ for $(l, l') \in \hookrightarrow$, and call the elements of $\hookrightarrow$ *edges*), label is a function that assigns a command to each edge, $\bot$ is the *start location*, and $\top$ is the *end location*. The following standard conditions must hold: (*i*) the only outgoing edge of $\top$ is $\top \hookrightarrow \top$; (*ii*) either all or none of the outgoing edges of a location are labeled by conditional statements; if all, then every configuration satisfies the condition of exactly one outgoing edge; if none, then the location has exactly one outgoing edge; (*iii*) if an outgoing edge of a location is labeled by an assignment, then it is the only outgoing edge of this location. A location is *nondeterministic* if it has an outgoing edge labeled by a nondeterministic assignment, otherwise it is *deterministic*. Deterministic locations can be *probabilistic* or *nonprobabilistic*. A program is deterministic if all its locations are deterministic.

Program Semantics. The semantics of a probabilistic program is an MDP. Let P be a *probabilistic program* $(\mathcal{L}, I, \hookrightarrow, \text{label}, \bot, \top)$, and let $\mathcal{L}_D, \mathcal{L}_A$ denote the sets of deterministic and nondeterministic locations of P. The semantics of P is the MDP $\mathcal{M}_P := (Q_A, Q_D, \text{Init}, \to, \text{Lab}_A, \text{Lab}_P)$, where $Q_A = \mathcal{L}_A \times \mathit{Val}$ is the set of nondeterministic nodes, $Q_D = ((\mathcal{L} \setminus \mathcal{L}_A) \times \mathit{Val}) \cup \{\top\}$ is the set of deterministic nodes, $\text{Init} = \{\bot\} \times I$ is the set of initial nodes, $\text{Lab}_A = \{a_0, a_1\}$ is the set of action labels, $\text{Lab}_P = \{\tau, 0, 1\}$ is the set of probabilistic labels, and the relation $\to$ is defined as follows: For every node $v = \langle l, \sigma \rangle$ of $\mathcal{M}_P$ and every edge $l \hookrightarrow l'$ of P

- if $\text{label}(l, l') = (x := \text{coin}(p))$, then $v \xrightarrow{0,p} \langle l', \sigma[x \mapsto 0] \rangle$ and $v \xrightarrow{1,1-p} \langle l', \sigma[x \mapsto 1] \rangle$;
- if $\text{label}(l, l') = (x := \text{nondet}())$, then $v \xrightarrow{a_0} \langle l', \sigma[x \mapsto 0] \rangle$ and $v \xrightarrow{a_1} \langle l', \sigma[x \mapsto 1] \rangle$;
- if $\text{label}(l, l') = (x := e)$, then $v \xrightarrow{\tau,1} \langle l', \sigma[x \to e(\sigma)] \rangle$, where $\sigma[x \to e(\sigma)]$ denotes the configuration obtained from σ by updating the value of x to the expression e evaluated under σ;
- if $\text{label}(l, l') = c$ for a conditional c satisfying σ, then $v \xrightarrow{\tau,1} \langle l', \sigma \rangle$.

For each node $v = \langle \top, \sigma \rangle$, $v \xrightarrow{\tau} \top$ and $\top \xrightarrow{\tau} \top$. □

A program $P = (\mathcal{L}, I, \hookrightarrow, \text{label}, \bot, \top)$ is called

- *a.s.-terminating* if $\Pr_q^S[\{r \in \text{Runs}(\mathcal{M}_P) \mid r \text{ reaches } \top\}] = 1$ for every strategy S and every initial state q of $\mathcal{M}_P$;
- *finite* if finitely many nodes are reachable from the initial nodes of $\mathcal{M}_P$;
- *weakly finite* if P_b is finite for all $b \in I$, where P_b is obtained from P by fixing b as the only initial node.

Assumption. We assume in the following that programs to be analyzed are deterministic. We consider nondeterministic programs only in Section 3.1.

3 Patterns

We introduce the notion of patterns for probabilistic programs. A pattern restricts a probabilistic program by imposing particular sequences of coin toss outcomes on the program runs. For the rest of the section we fix a probabilistic program $P = (\mathcal{L}, I, \hookrightarrow, \text{label}, \bot, \top)$ and its associated MDP $\mathcal{M}_P = (Q_A, Q_P, \text{Init}, \to, \text{Lab}_A, \text{Lab}_P)$.

We write $C := \{0, 1\}$ for the set of coin toss outcomes in the following. A *pattern* is a subset of C^ω of the form $C^* w_1 C^* w_2 C^* w_3 \ldots$, where $w_1, w_2, \ldots \in \Sigma^*$. We say the sequence $w_1, w_2, \ldots$ *induces* the pattern. Fixing an enumeration $x_1, x_2, \ldots$ of C^*, we call the pattern induced by $x_1, x_2, \ldots$ the *universal* pattern. For a pattern Φ, a run $r \in \text{Runs}(\mathcal{M}_P)$ is *Φ-conforming* if there is $v \in \Phi$ such that $\bar{r}|_C$ is a prefix of v. We call a pattern Φ *terminating (for P)* if all Φ-conforming runs terminate, i.e., reach $\top$. We show the following theorem:

Theorem 2.

(1) Let Φ be a pattern. The set of Φ-conforming runs has probability 1*. In particular, if Φ is terminating, then P is a.s.-terminating.*
(2) If P is a.s.-terminating and weakly finite, then the universal pattern is terminating for P.
(3) If P is a.s.-terminating and finite with $n < \infty$ reachable nodes in $\mathcal{M}_P$, then there exists a word $w \in C^*$ with $|w| \in \mathcal{O}(n^2)$ such that $C^* w C^\omega$ is terminating for P.

Part (1) of Theorem 2 is the basis for the pattern approach. It allows to ignore runs that are not Φ-conforming, because they have probability 0. Part (2) states that the pattern approach is "complete" for a.s.-termination and weakly finite programs: For any a.s.-terminating and weakly finite program there is a terminating pattern; moreover the universal pattern suffices. Part (3) refines part (2) for finite programs: there is a short word such that $C^* w C^\omega$ is terminating.

Proof (of Theorem 2).
Part (1) (Sketch): We can show that the set of runs r that visit infinitely many probabilistic nodes and do not have the form $C^* w_1 C^\omega$ is a null set. This result can then easily be generalized to $C^* w_1 C^* w_2 \ldots C^* w_n C^\omega$. All runs conforming Φ can then be formed as a countable intersection of such run sets.

Part (2): Let $\sigma_1, \sigma_2, \ldots$ be a (countable or infinite) enumeration of the nodes in I. With Part (3) we obtain for each $i \geq 1$ a word w_i such that $C^* w_i C^\omega$ is a terminating pattern for P, if the only starting node considered is σ_i. By its definition, the universal pattern is a subset of $C^* w_i C^\omega$ for every $i \geq 1$, so it is also terminating.

Part (3) (Sketch): Since P is a.s.-terminating, for every node q there exists a coin toss sequence w_q, $|w_q| \leq n$, with the following property: a run that passes

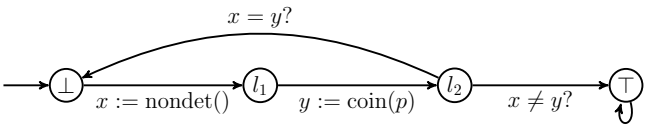

Fig. 4. Nondeterministic a.s.-terminating program without terminating pattern

through q and afterwards visits exactly the sequence w_q of coin toss outcomes is terminating. We build a sequence w such that for every state q every run that passes through q and then visits exactly the sequence w is terminating. We start with $w = w_q$ for an arbitrary $q \neq \top$. Then we pick a $q' \neq \top$ such that for $q'' \neq q$, runs starting in q'' and visiting exactly the probabilistic label sequence w lead to q'. We set $w = w_q w_{q'}$; after visiting w, all runs starting from q and q' end in $\top$. We iterate this until no more q' can be found. We stop after at most n steps and obtain a sequence w of length $\leq n^2$. □

3.1 Nondeterministic Programs

For nondeterministic a.s.-terminating programs, there might not exist a terminating pattern, even if the program is finite. Figure 4 shows an example. Let Φ be a pattern and $c_1 c_2 c_3 \ldots \in \Phi$. The run

$$\langle \bot, \sigma_0 \rangle \xrightarrow{a_{c_1}} \langle l_1, \sigma_1 \rangle \xrightarrow{c_1} \langle l_2, \sigma_1' \rangle \xrightarrow{\tau} \langle \bot, \sigma_1' \rangle \xrightarrow{a_{c_2}} \langle l_1, \sigma_2 \rangle \xrightarrow{c_2} \langle l_2, \sigma_2' \rangle \xrightarrow{\tau} \langle \bot, \sigma_2' \rangle \xrightarrow{a_{c_3}} \ldots$$

in $\mathcal{M}_P$ is Φ-conforming but nonterminating.

We show that the concept of patterns can be suitably generalized to nondeterministic programs, recovering a close analog of Theorem 2. Assume that the program is in a normal form where nondeterministic and probabilistic locations strictly alternate. This is easily achieved by adding dummy assignments. Writing $A := \{a_0, a_1\}$, every run $r \in \mathcal{M}_P$ satisfies $r|_{A \cup C} \in (AC)^\infty$.

A *response* of length n encodes a mapping $A^n \to C^n$ in an "interleaved" fashion, e.g., $\{a_0 1, a_1 0\}$ is a response of length one, $\{a_0 0 a_0 1, a_0 0 a_1 1, a_1 0 a_0 1, a_1 0 a_1 1\}$ is a response of length two. A *response pattern* is a subset of $(AC)^\omega$ of the form $(AC)^* R_1 (AC)^* R_2 (AC)^* \ldots$, where $R_1, R_2, \ldots$ are responses. If we now define the notions of *universal* and *terminating* response patterns analogously to the deterministic case, a theorem very much like Theorem 2 can be shown. For instance, let $\Phi = (AC)^* \{a_0 1, a_1 0\} (AC)^\omega$. Then every Φ-conforming run of the program in Fig. 4 has the form

$$\langle \bot, \sigma_0 \rangle \to \ldots \to q \xrightarrow{a_i} q' \xrightarrow{1-i} q'' \to \top \to \ldots \qquad \text{for an } i \in \{0, 1\}.$$

This implies that the program is a.s.-terminating. See [6] for the details.

4 Our Algorithm

In this section we aim at a procedure that, given a weakly finite program P, proves that P is a.s.-terminating by computing a terminating pattern. This

approach is justified by Theorem 2 (1). In fact, the proof of Theorem 2 (3) constructs, for any finite a.s.-terminating program, a terminating pattern. However, the construction operates on the Markov chain $\mathcal{M}_P$, which is expensive to compute. To avoid this, we would like to devise a procedure which operates on P, utilizing (nonprobabilistic) verification tools, such as model checkers and termination provers.

Theorem 2 (2) guarantees that, for any weakly finite a.s.-terminating program, the universal pattern is terminating. This suggests the following method for proving a.s.-termination of P: (i) replace in P all probabilistic assignments by nondeterministic ones and instrument the program so that all its runs are conforming to the universal pattern (this can be done as we describe in Section 4.1 below); then (ii) check the resulting program for termination with a termination checker such as ARMC [18]. Although this approach is sound and complete (modulo the strength of the termination checker), it turns out to be useless in practice. This is because the crucial loop invariants are extremely hard to catch for termination checkers. Already the instrumentation that produces the enumeration of C^* requires a nontrivial procedure (such as a binary counter) whose loops are difficult to analyze.

Therefore we devise in the following another algorithm which tries to compute a terminating pattern $C^*w_1C^*w_2\ldots$ It operates on P and is "refinement"-based. Our algorithm uses a "pattern checker" subroutine which takes a sequence $w_1, w_2, \ldots$, and checks (or attempts to check) whether the induced pattern is terminating. If it is not, the pattern checker may return a *lasso* as counterexample. Formally, a lasso is a sequence

$$\langle l_1,\sigma_1\rangle \to \langle l_2,\sigma_2\rangle \to \ldots \to \langle l_m,\sigma_m\rangle \to \ldots \to \langle l_n,\sigma_n\rangle \quad \text{with } \langle l_n,\sigma_n\rangle \to \langle l_m,\sigma_m\rangle$$

and $\langle l_1,\sigma_1\rangle \in \text{Init}$. We call the sequence $\langle l_m,\sigma_m\rangle \to \ldots \to \langle l_n,\sigma_n\rangle$ the *lasso loop* of the lasso. Note that a lasso naturally induces a run in $\text{Runs}(\mathcal{M}_P)$. If P is finite, pattern checkers can be made complete, i.e., they either prove the pattern terminating or return a lasso.

We present our pattern-finding algorithms for finite-state and weakly finite programs. In Section 4.1 we describe how pattern-finding and pattern-checking can be implemented using existing verification tools.

Finite Programs. First we assume that the given program P is finite. The algorithm may take a *base word* $s_0 \in C^*$ as input, which is set to $s_0 = \epsilon$ by default. Then it runs the pattern checker on $C^*s_0C^*s_0\ldots$ If the pattern checker shows the pattern terminating, then, by Theorem 2 (1), P is a.s.-terminating. Otherwise the pattern checker provides a lasso $\langle l_1,\sigma_1\rangle \to \ldots \to \langle l_m,\sigma_m\rangle \to \ldots \to \langle l_n,\sigma_n\rangle$. Our algorithm extracts from the lasso loop a word $u_1 \in C^*$, which indicates a sequence of outcomes of the coin tosses in the lasso loop. If $u_1 = \epsilon$, then the pattern checker has found a nonterminating run with only finitely many coin tosses, hence P is not a.s.-terminating. Otherwise (i.e., $u_1 \neq \epsilon$), let $s_1 \in C^*$ be a shortest word such that s_0 is a prefix of s_1 and s_1 is not an infix of u_1^ω. Our algorithm runs the pattern checker on $C^*s_1C^*s_1\ldots$ If the pattern checker shows the pattern terminating, then P is a.s.-terminating. Otherwise

the pattern checker provides another lasso, from which our algorithm extracts a word $u_2 \in C^*$ similarly as before. If $u_2 = \epsilon$, then P is not a.s.-terminating. Otherwise, let $s_2 \in C^*$ be a shortest word such that s_0 is a prefix of s_2 and s_2 is neither an infix of u_1^ω nor an infix of u_2^ω. Observe that the word s_1 is an infix of u_2^ω by construction, hence $s_2 \neq s_1$. Our algorithm runs the pattern checker on $C^* s_2 C^* s_2 \dots$ and continues similarly. More precisely, in the $i-th$ iteration it chooses s_i as a shortest word such that s_i is a prefix of s_{i-1} and s_i is not an infix of any of the words $u_1^\omega, \dots, u_i^\omega$, thus eliminating all lassos so far discovered.

The algorithm is complete for finite and a.s.-terminating programs:

Proposition 3. *Let P be finite and a.s.-terminating. Then the algorithm finds a shortest word w such that the pattern $C^* w C^* w \dots$ is terminating, thus proving termination of P.*

In each iteration the algorithm picks a word s_j that destroys all previously discovered lasso loops. If the loops are small, then the word is short:

Proposition 4. *We have $|s_j| \leq |s_0| + 1 + \log_2 (|u_1| + \dots + |u_j|)$.*

The proofs for both propositions can be found in [6].

Weakly Finite Programs. Let us now assume that P is a.s.-terminating and weakly finite. We modify our algorithm. Let $b_1, b_2, \dots$ be an enumeration of the set I of initial nodes. Our algorithm first fixes b_1 as the only initial node. This leads to a finite program, so we can run the previously described algorithm, yielding a word w_1 such that $C^* w_1 C^* w_1 \dots$ is terminating for the initial node b_1. Next our algorithm fixes b_2 as the only initial node, and runs the previously described algorithm taking w_1 as base word. As before, this establishes a terminating pattern $C^* w_2 C^* w_2 \dots$ By construction of w_2, the word w_1 is a prefix of w_2, so the pattern $C^* w_1 C^* w_2 C^* w_2 \dots$ is terminating for the initial nodes $\{b_1, b_2\}$. Continuing in this way we obtain a sequence $w_1, w_2, \dots$ such that $C^* w_1 C^* w_2 \dots$ is terminating. Our algorithm may not terminate, because it may keep computing $w_1, w_2, \dots$. However, we will illustrate that it is promising to compute the first few w_i and then *guess* an expression for general w_i. For instance if $w_1 = 0$ and $w_2 = 00$, then one may guess $w_i = 0^i$. We encode the guessed sequence $w_1, w_2, \dots$ in a finite way and pass the obtained pattern $C^* w_1 C^* w_2 \dots$ to a pattern checker, which may show the pattern terminating, establishing a.s.-termination of the weakly finite program P.

4.1 Implementing Pattern Checkers

Finite Programs. We describe how to build a pattern checker for finite programs P and patterns of the form $C^* w C^* w \dots$ We employ a model checker for finite-state nonprobabilistic programs that can verify temporal properties: Given as input a finite program and a Büchi automaton $\mathcal{A}$, the model checker returns a lasso if there is a program run accepted by $\mathcal{A}$ (such runs are called "counterexamples" in classical terminology). Otherwise it states that there is no counterexample. For our case studies, we use the SPIN tool [9].

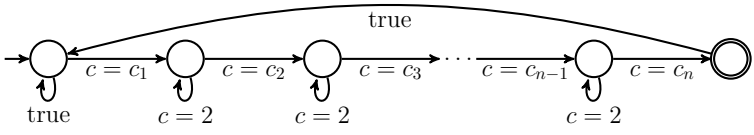

Fig. 5. Büchi automaton $\mathcal{A}(w)$, for $w = c_1 c_2 \dots c_n \in C^*$. Note that the number of states in $\mathcal{A}(w)$ grows linearly in $|w|$.

Given a finite probabilistic program P and a pattern $\Phi = C^* w C^* w \dots$, we first transform P into a nonprobabilistic program P' as follows. We introduce two fresh variables c and term, with ranges $\{0, 1, 2\}$ and $\{0, 1\}$, respectively, and add assignments `term = 0` and `term = 1` at the beginning and end of the program, respectively. Then every location l of P with $label(l, l') = x = \text{coin}(p)$ for a label l' is replaced by a nondeterministic choice and an if-statement as follows:

```
x = nondet();
if (x = 0)  { c = 0; c = 2 } else { c = 1; c = 2 } end if;
```

In this way we can distinguish coin toss outcomes in a program trace by inspecting the assignments to c. Now we perform two checks on the nonprobabilistic program P': First, we use SPIN to translate the LTL formula $G\,\neg\text{term} \wedge FG(c \notin \{0, 1\})$ into a Büchi automaton and check whether P' has a run that satisfies this formula. If there is indeed a lasso, our pattern checker reports it. Observe that by the construction of the LTL formula the lasso encodes a nonterminating run in P that eventually stops visiting probabilistic locations. So the lasso loop does not contain any coin tosses (and our algorithm will later correctly report that P is not a.s.-terminating). Otherwise, i.e., if no run satisfies the formula, we know that all nonterminating runs involve infinitely many coin tosses. Then we perform a second query: We construct a Büchi automaton $\mathcal{A}(w)$ that represents the set of infinite Φ-conforming runs, see Fig. 5. We use SPIN to check whether P' has run that is accepted by $\mathcal{A}(w)$. If yes, then there is an infinite Φ-conforming run, and our pattern checker reports the lasso. Otherwise, it reports that Φ is a terminating pattern.

Weakly Finite Programs. Recall that for weakly finite programs, the pattern checker needs to handle patterns of a more general form, namely $\Phi = C^* w_1 C^* w_2 \dots$ Even simple patterns like $C^* 0 C^* 00 C^* 000 \dots$ cannot be represented by a finite Büchi automaton. Therefore we need a more involved instrumentation of the program to restrict its runs to Φ-conforming ones. Now our pattern checker employs a termination checker for infinite-state programs. For our experiments we use ARMC.

Given a weakly finite program P and a pattern $\Phi = C^* w_1 C^* w_2 \dots$, we transform P into a nonprobabilistic program P^Φ as follows. We will use a command `x = ?`, which nondeterministically assigns a nonnegative integer to x. Further we assume that we can access the k-th letter of the i-th element of $(w_i)_{i \in \mathbb{N}}$ by w[i][k], and $|w_i|$ by length(w[i]). We add fresh variables `ctr`, `next` and `pos`, where `ctr` is initialized nondeterministically with any nonnegative integer and

```
x = nondet();
if (ctr <= 0)
  if (pos > length(w[next])) { ctr = ?; pos = 1; next = next+1 }
  else { x = w[next][pos]; pos = pos+1 }
else ctr = ctr-1
```

Fig. 6. Code transformation for coin tosses in weakly finite programs

`next` and `pos` are both initialized with 1. If a run r is Φ-conforming, $\bar{r}|_C$ is a prefix of $v_1 w_1 v_2 w_2 v_3 w_3 \ldots$, with $v_i \in C^*$. The variable `ctr` is used to "guess" the length of the words v_i; the individual letters in v_i are irrelevant. We replace every command $c = \mathrm{coin}(p)$ by the code sequence given in Fig. 6.

The runs in the resulting program P^Φ correspond exactly to the Φ-conforming runs in P. Then P^Φ is given to the termination checker. If it proves termination, we report "Φ is a terminating pattern for P". Otherwise, the tool might either return a lasso, which our pattern checker reports, or give up on P^Φ, in which case our pattern checker also has to give up.

In our experiments, a weakly finite program typically has an uninitialized integer variable N whose value is nondeterministically fixed in the beginning. The pattern $C^* w_1 C^* \ldots C^* w_N C^\omega$ is then often terminating, which makes $\texttt{next} \leq N$ an invariant in P^Φ. The termination checker ARMC may benefit from this invariant, but may not be able to find it automatically (for reasons unknown to the authors). We therefore enhanced ARMC to "help itself" by adding the invariant next $\leq N$ to the program if ARMC's reachability mode can verify the invariant.

5 Experimental Evaluation

We apply our methods to several parameterized programs taken from the literature.[1]

- *firewire:* Fragment of FireWire's symmetry-breaking protocol, adapted from [12] (a simpler version was used in the introduction). Roughly speaking, the number 100 of Fig. 1 is replaced by a parameter N.
- *randomwalk:* A slightly different version of the finite-range, one-dimensional random walk used as second example in the introduction.
- *herman:* An abstraction of Herman's randomized algorithm for leader election used in [14]. It can be seen as a more complicated finite random walk, with N as the walk's length.
- *zeroconf:* A model of the Zeroconf protocol taken from [10]. The protocol assigns IP addresses in a network. The parameter N is the number of probes sent after choosing an IP address to check whether it is already in use.
- *brp:* A model adapted from [10] that models the well-known bounded retransmission protocol. The original version can be proven a.s.-terminating with the trivial pattern C^ω; hence we study an "unbounded" version, where arbitrarily many retransmissions are allowed. The parameter N is the length of the message that the sender must transmit to the receiver.

[1] The sources can be found at `http://www.model.in.tum.de/~gaiser/cav2012.html`

Name	#loc	Pattern words for $N = 1, 2, 3, 4$				Time (SPIN)	i-th word of guessed pattern	Time (ARMC)
firewire	19	010	010	010	010	17 sec	010	1 min 36 sec
randomwalk	16	ϵ	0^2	0^3	0^4	23 sec	0^i	1 min 22 sec
herman	36	010	$0(10)^2$	$0(10)^3$	$0(10)^4$	47 sec	$0(10)^i$	7 min 43 sec
zeroconf	39	0^3	0^4	0^5	0^6	20 sec	0^{i+2}	26 min 16 sec
brp	57	00	00	00	00	19 sec	00	45 min 14 sec

Fig. 7. Constructed patterns of the case studies and runtimes

Proving a.s.-termination. We prove a.s.-termination of the examples using SPIN [9] to find patterns of finite-state instances, and ARMC [18] to prove termination of the nondeterministic programs derived from the guessed pattern. All experiments were performed on an Intel© i7 machine with 8GB RAM. The results are shown in Fig. 7. The first two columns give the name of the example and its size. The next two columns show the words $w_1, \ldots, w_4$ of the terminating patterns $C^*w_1C^\omega, \ldots, C^*w_4C^\omega$ computed for $N = 1, 2, 3, 4$ (see Theorem 2(3) and Section 4.1), and SPIN's runtime. The last two columns give word w_i in the guessed pattern $C^*w_1C^*w_2C^*w_3\ldots$ (see Section 4.1), and ARMC's runtime. For instance, the entry $0(10)^i$ for *herman* indicates that the guessed pattern is $C^*010C^*01010C^*0101010\ldots$.

We derive two conclusions. First, a.s.-termination is proved by very simple patterns: the general shape is easily guessed from patterns for $N = 1, 2, 3, 4$, and the need for human ingenuity is virtually reduced to zero. This speaks in favor of the Planner technique of [1] and our extension to patterns, compared to other approaches using fairness and Hoare calculus [16,11]. Second, the runtime is dominated by the termination tool, not by the finite-state checker. So the most direct way to improve the efficiency of our technique is to produce faster termination checkers.

In the introduction we claimed that general purpose probabilistic model-checkers perform poorly for a.s.-termination, since they are not geared towards this problem. To supply some evidence for this, we tried to prove a.s.-termination of the first four examples using the CEGAR-based PASS model checker [7,8]. In all four cases the refinement loop did not terminate.[2]

Improving Lower Bounds for Reachability. Consider a program of the form `if coin(0.8) P1() else P2(); ERROR` . Probabilistic model-checkers compute lower and upper bounds for the probability of ERROR. Loosely speaking, lower bounds are computed by adding the probabilities of terminating runs of P1 and P2. However, since CEGAR-based checkers [7,8,10,5] work with abstractions of P1 and P2, they may not be able to ascertain that paths of the abstraction are concrete paths of the program, leading to poor lower bounds. Information on a.s.-termination helps: if e.g. P1 terminates a.s., then we already have a lower

[2] Other checkers, like PRISM, cannot be applied because they only work for finite-state systems.

bound of 0.8. We demonstrate this technique on two examples. The first one is the following modification of *firewire*:

```
N = 1000; k = 0; miss = 0;
while (k < N) {
  old_x = x; x = coin(0.5);
  if (x = old_x) k++ else if (k < 5) miss = 1
}
```

For $i \in \{0, 1\}$, let p_i be the probability that the program terminates with miss $= i$. After 20 refinement steps PASS returns upper bounds of 0.032 for p_0 and 0.969 for p_1, but a lower bound of 0 for p_1, which stays 0 after 300 iterations. Our algorithm establishes that the loop a.s.-terminates, which implies $p_0 + p_1 = 1$, and so after 20 iterations we already get $0.968 \leq p_1 \leq 0.969$.

We apply the same technique to estimate the probabilities p_1, p_0 that *zeroconf* detects/does-not-detect an unused IP address. For $N = 100$, after 20 refinement steps PASS reports an upper bound of 0.999 for p_0, but a lower bound of 0 for p_1, which stays 0 for 80 more iterations. With our technique after 20 iterations we get $0.958 \leq p_1 \leq 0.999$.

6 Conclusions

We have presented an approach for automatically proving a.s.-termination of probabilistic programs. Inspired by the Planner approach of [1], we instrument a probabilistic program P into a nondeterministic program P' such that the runs of P' correspond to a set of runs of P with probability 1. The instrumentation is fully automatic for finite-state programs, and requires an extrapolation step for weakly finite programs. We automatically check termination of P' profiting from new tools that were not available to [1]. While our approach maintains the intuitive appeal of the Planner approach, it allows to prove completeness results. Furthermore, while in [1] the design of the Planner was left to the verifier, we have provided in our paper a CEGAR-like approach. In the case of parameterized programs, the approach requires an extrapolation step, which however in our case studies proved to be straightforward. Finally, we have also shown that our approach to improve the game-based CEGAR technique of [7,8,10] for computing upper and lower bounds for the probability of reaching a program location. While this technique often provides good upper bounds, the lower bounds are not so satisfactory (often 0), due to spurious nonterminating runs introduced by the abstraction. Our approach allows to remove the effect of these runs.

In future work we plan to apply learning techniques to pattern generation, thereby inferring probabilistic termination arguments for large program instances from small instances.

Acknowledgments. We thank the referees for helping us clarify certain aspects of the paper, Corneliu Popeea and Andrey Rybalchenko for many discussions and their help with ARMC, and Björn Wachter and Florian Zuleger for fruitful insights on quantitative probabilistic analysis and termination techniques.

References

1. Arons, T., Pnueli, A., Zuck, L.D.: Parameterized Verification by Probabilistic Abstraction. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 87–102. Springer, Heidelberg (2003)
2. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)
3. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM 32, 824–840 (1985)
4. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond Safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
5. Esparza, J., Gaiser, A.: Probabilistic Abstractions with Arbitrary Domains. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 334–350. Springer, Heidelberg (2011)
6. Esparza, J., Gaiser, A., Kiefer, S.: Proving termination of probabilistic programs using patterns. Technical report (2012), http://arxiv.org/abs/1204.2932
7. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: Abstraction Refinement for Infinite Probabilistic Models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)
8. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
9. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional (2003)
10. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Abstraction Refinement for Probabilistic Software. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
11. McIver, A., Morgan, C.: Developing and Reasoning About Probabilistic Programs in *pGCL*. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 123–155. Springer, Heidelberg (2006)
12. McIver, A., Morgan, C., Hoang, T.S.: Probabilistic Termination in *B*. In: Bert, D., Bowen, J. P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 216–239. Springer, Heidelberg (2003)
13. Monniaux, D.: An Abstract Analysis of the Probabilistic Termination of Programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 111–126. Springer, Heidelberg (2001)
14. Nakata, T.: On the expected time for Herman's probabilistic self-stabilizing algorithm. Theoretical Computer Science 349(3), 475–483 (2005)
15. Pnueli, A.: On the extremely fair treatment of probabilistic algorithms. In: STOC, pp. 278–290. ACM (1983)
16. Pnueli, A., Zuck, L.D.: Probabilistic verification. Inf. Comput. 103, 1–29 (1993)
17. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society (2004)
18. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2006)
19. Podelski, A., Rybalchenko, A.: Transition Invariants and Transition Predicate Abstraction for Program Termination. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 3–10. Springer, Heidelberg (2011)
20. Rybalchenko, A.: Temporal verification with transition invariants. PhD thesis (2005)