Rheinisch-Westfälisch Technische Hochschule Aachen

Seminar Paper

Proving Termination of Probabilistic Programs using Patterns

Seminar *Probabilistic Programs* – Winter Term 2014/2015

Chair i2 – Software Modelling and Verification

Abstract

The paper presents an algorithm that proves termination of probabilistic programs. This issue is reducible to the halting problem, which is undecidable. Thus, the algorithm will make guesses and won't guarantee completeness and correctness for all possible programs at the same time. We will introduce the notion of "almost sure termination" and devise an algorithm that uses a refinement-based approach to prove whether a probabilistic program terminates with the likelihood of one. For this, patterns of possible probabilistic choices are constructed by the algorithm. A pattern indicates special properties of probabilistic runs, namely it defines a family of runs for which a structural representation of coin-toss outcomes holds. The algorithm then tries to construct a pattern whose induced runs are terminating.

Author: Lukas Westhofen *Supervisor:* Nils Jansen

1 Introduction

Probabilistic programs are used in a wide variety of fields, including probabilistic network protocols, machine learning, biologic models and robotics. They allow for randomization of algorithms such that desired properties are potentially reached faster. For example, the well-known FireWire-protocol, which is explained in [3], implements such a probabilistic algorithm that solves the problem of electing a root node in its hierarchical tree-like network structure. Roughly speaking, every node sends *"Be my parent node"* to other nodes to iteratively construct a tree-structure. If neighbored nodes both receive this message from each other, a conflict is risen. This clash is then solved by both nodes flipping a coin until they show a different outcome. The node whose coin shows head is then elected as the parent node. But a problem arises: Does the process of coin-flipping ever end? If both nodes always produce the same coin-toss outcomes, the election will fail and ultimately the protocol communication will come to a halt. Thus, we need to prove termination of probabilistic programs in order to guarantee the functionality of developed protocols and algorithms.

In this paper, which is based on [5], an algorithm is devised which proves termination of such processes. For this, we examine probabilistic programs on their property of halting with probability one. Intuitively, this is the case if the set of all terminating runs has probability one – Thus, the program is assured to be terminating with probability one.

Let us examine an example probabilistic program to clarify the notion of "terminating runs with probability one" and the later presented approach on termination checking.

```
1 k = 0;
2 x = 0;
3 while(k < 10) {
4     old_x = x;
5     x = coin(0.5)
6     if(x != old_x)
7         k = k + 1;
8 }
```

Figure 1: Textual representation of a simple probabilistic program that terminates almost surely.

In Code Snippet 1 a simple probabilistic program is shown. Note that it has no parameters at all. Similar to [5], we denote `coin(p)` as a function that returns with probability p one and with probability $1 - p$ zero. We set $0 < p < 1$. Loosely speaking, `coin(p)` flips a coin that may be unfair, thus having different probabilities for each side of the coin. In this example, we use a fair coin, which means every outcome has the same probability.

On the first sight, it may not be obvious whether that program terminates or not. Using human intuition, we note that the program terminates if the outcome of `coin(0.5)` alternates between 1 and 0 ten times. Also, if we keep tossing that coin over a very long time period, stochastic theory guarantees us that we will somewhen see such a finite alternation between both outcomes. So we observe, via intuition, that Program 1 terminates. But there exist also non-terminating runs, such as the run where `coin(0.5)` returns 0 at any point. But since the probability of this run is 0 – because we will eventually see a 1-result in every run – we disregard such runs in our termination-analysis.

To formalize this intuitive stochastic thinking the term *almost sure termination*, which defines a property on probabilistic programs, is used in [5]. This notion is known from stochastic theory, where an event is called to happen almost surely if it has the probability 1 [1, p.176]. We say that a given probabilistic program terminates almost surely if the probability of all terminating runs equals 1. A run is a specific order of execution states in which variables and their values are specified. For example, the run where `coin(p)` yields at first 0 and afterwards 10 times 1 is a terminating run. We will see that a run is exactly defined by the outcomes of the `coin`-calls if we fix possible input parameters.

Going back to the above example, we can see that all runs containing ten alternations between the coin-tosses terminate. This induces a *pattern* $\phi \subseteq \{0,1\}^\omega$ that outlines the shape of every

terminating run via its associated coin toss outcomes: $\phi = (C^*10)^{10}C^\omega$. We will characterize patterns using infinite concatenations of regular expressions over finite languages. The presented algorithm will use those patterns to prove termination by constructing such a pattern iteratively and employing hereafter mentioned model checkers to prove that all runs induced by this pattern are terminating ones.

Before we examine the presented termination checker, we need to take a glance at the *halting problem*. The halting problem states a fundamental challenge to computer scientists. Intuitively, it asks for an algorithm that decides whether a given program halts on a specific input.

Definition 1 (Halting problem, [4, p.70])**.** *Find a Turing machine M such that, for any other Turing machine P and an input i, M returns "1" if and only if P terminates after finitely many computation steps on i.*

Since Alan Turing showed the undecidability of the halting problem in 1936, we should handle the topic of proving program termination with care. We recognize that any attempt to create a correct and complete termination prover will ultimately fail. This effects the content of this paper drastically: Although an algorithm will be presented that tries to prove termination of probabilistic programs, one should keep the undecidability in mind. But, on the bright side, we can reduce the question of termination to certain subclasses of programs – We can, for example, prove termination of loop- and recursion-free programs since they do not allow for infinite runs due to their structure. Alternatively, we are able create algorithms that at least for quite a lot of cases return an answer, but can also return "maybe" or loop to infinity for some inputs.

Computer scientists are now able to reduce the question of termination to certain subclasses of programs and by now, there exist algorithms that are quite often able to tell whether a program halts or not. For this purpose, we can use a so called *model-checker*, since we are able to express the halting property in such a way that a model checker is hopefully able to verify it. In general, a model checker answers the question:

> Does a given property p hold on a given formal system M? [2, p.13]

In other words, we are looking for the truth content of the relation $M \models p$. We can employ those tools for proving termination via asking if a given program P fulfills the LTL-formula $GF\top$, which states that for every program state there needs to be a program state later on that indicates the "end" position (called $\top$ in the formula). Since most model-checkers are able to verify LTL-properties of systems, those tools can be used to attempt a termination proof.

SPIN might be one of the best-known model-checkers. SPIN works on Finite State Machines (FSMs) and LTL-formulas, a special temporal logic [8]. We express our FSM-property in such an LTL-formula φ (a so-called *never-claim*) and, along with the FSM A that is to be evaluated, pass it to SPIN, which then returns True if φ holds for every run of the system A, i.e. $A \models \varphi$. Otherwise we are provided with a counterexample. A counterexample is a run of A for which φ does not hold. In this paper, SPIN will be used to verify temporal properties of non-probabilistic non-deterministic programs. Since we work on probabilistic programs, the presented algorithm needs to transform those into non-probabilistic programs for which SPIN is then able to verify properties on.

ARMC is a more specialized tool than SPIN – It verifies termination and reachability properties of a given program [9]. Internally, it uses the logical programming language Prolog. Similar to SPIN, we will not run ARMC on probabilistic programs directly, but on non-probabilistic non-deterministic ones. The advantage of ARMC over SPIN is that the input program is not limited to be finite-state, which i.a. allows for parameters.

2 Fundamentals

2.1 Probabilistic programs

A program itself is, colloquially, a finite collection of instructions for a computational machine. We will not further specify whether a program needs to be deterministic or non-deterministic, since our algorithm will nevertheless perform a non-deterministic transformation of its input program. A probabilistic program introduces randomness – We will allow the machine, with a given probability p and based on a discrete stochastic distribution, to choose one of several alternative instruction sequences.

In this paper, a probabilistic program and their variables will be defined to only work on integers as values. Additionally, our textual representation of control structures will look common to the *C* programming language. For every program, we will allow an arbitrary number of parameters which can be named as desired.

2.1.1 Syntax

Definition 2 (Parts of probabilistic programs)**.**

- Var *is the set of all possible variables over the alphabet of digits and standard letters.*
- Conf *is the countable set of all possible configurations of Var: For $c \in Conf$ it holds that c is a function such that $c : Var \to \mathbb{Z}$.*
- *We introduce* `<`, `<=`, `==` *and* `!=` *as relations between variables and integers. We call the resulting terms* expressions*. Every boolean combination of expressions is again an expression itself.*
- *As* commands*, we allow*
  - assignments*, where for every $v, v' \in Var$ and $i \in \mathbb{Z}$,* `v = i` *and* `v = v'` *is an assignment. The right side of an assignment is allowed to include arithmetics over $+, -$ and $\cdot$. Additionally, for the sake of simplicity, we permit assignments of expressions to variables, so for every expression e and $v \in Var$,* `v = e` *is an assignment.*
  - conditions*, where for every expression e* `if(e)` *is a condition.*

And since we want to incorporate random choices into our programs, we need our priorly mentioned `coin`*-method:*

- *We introduce* `coin(p)` *for $0 < p < 1$ as an expression. This allows us to use the* `coin`*-method in both assignments and conditions.*

For non-deterministic probabilistic programs we need to define special assignments to allow non-deterministic configuration-alternation.

- *We add* `nondet()` *to our set of expressions.* `nondet()` *will non-deterministically return a 0 or 1.*

Finally, welding all those constructions together, we are able to define a probabilistic program syntactically by using a directed graph whose nodes represent, roughly speaking, the lines of a text-based program depiction.

Definition 3 (Probabilistic Programs)**.** *A* probabilistic program *P is a directed labeled graph where $P = (Locations, Init, \hookrightarrow, Labels, \bot, \top)$.*

- Locations $\subset \mathbb{N}$ *defines the finite set of control flow locations. Each command will lead to such a location. Further on we define the set of non-deterministic locations* $Locations_n \subseteq Locations$*, where for every location holds that it has at least one outgoing transition labeled with the expression* `nondet()`*. The set* $Locations_d = Locations \setminus Locations_n$ *denotes the deterministic locations.*

- Init $\subseteq$ *Conf is the set of initial configurations, i.e. the variable and parameter values that are set immediately before the program start.*

- $\hookrightarrow \subseteq (Locations \times Locations)$ *is the control flow relation. It connects locations via directed edges.*

- $\bot \in Locations$ *denotes the initial location, while* $\top \in Locations$ *indicates the termination location of the program.*

- Labels: $(\hookrightarrow \setminus (\top, \top)) \to Commands$ *assigns every label with exactly one command. We exclude the termination location for practical sakes.*

Further on we need to define some constraints regarding the conditional statements and the special nodes $\top$ and $\bot$.

Definition 4 (Constraints on probabilistic programs)**.**

1. *The only outgoing edge of* $\top$ *points to* $\top$ *itself.*
2. *All outgoing edges of a location must be of the same type: Either assignments or conditions.*
3. *If we label all outgoing edges of a location as conditions, we have to ensure that every possible program configuration satisfies exactly one of those conditions.*

The last condition is needed in order to ensure that a program does not unconsciously become non-deterministic – albeit we allow non-determinism in general via the `nondet()`-statement.

After all, we can represent a probabilistic program just as well in the already-known textual form. To understand the formal construction of probabilistic programs an example is given. Figure 2 shows the conversion from the before mentioned example program to its associated directed graph.

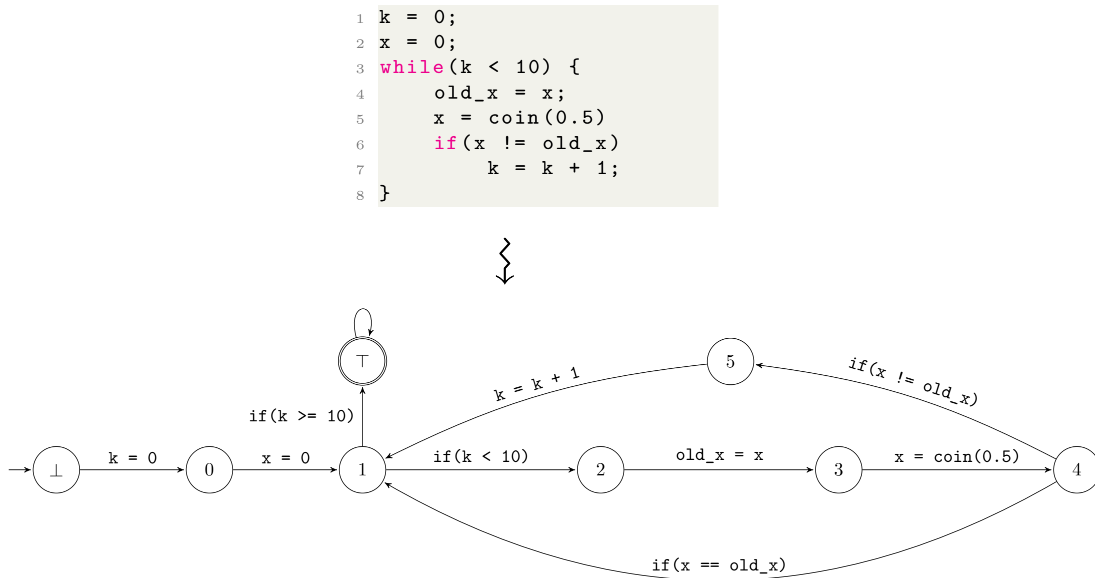

Figure 2: Above, a textual representation of Example 1 is shown, the associated directed graph is presented below.

2.1.2 Semantics

Once we defined the syntax of probabilistic programs, life needs to be breathed into those concepts via defining its operational semantics. In this section we explicate how to interpret above mentioned directed graph. We do so by using *Markov Decision Processes* (MDPs), a special class of stochastic systems.

MDPs provide two kinds of possible transitions: On one side we have the already-known *action transitions* which are similar to NFA-transitions and labeled with the possible actions the MDP can perform. On the other side we have *probabilistic transitions* which we also label, but additionally introduce a probability to each transition. The MDP then assures that the transition is taken with the given likelihood on the basis of a given discrete probability measure. Since MDPs allow for probabilistic choices in successor-node choosing, they fit our requirements for modeling probabilistic programs: We can utilize the action transitions to represent non-probabilistic program state changes (such as `x = nondet()`). For the probabilistic parts of our program we will use the probabilistic transitions: For every command that includes `coin(p)` we introduce two new transitions, one will lead with probability p to the program state where `coin(p)` returned 1, the other one will lead with probability $p - 1$ to the state where `coin(p)` yielded 0.

Definition 5 (Markov Decision Process)**.** *We define a Markov Decision Process* $\mathfrak{M}$ *as a tupel* $\mathfrak{M} = (Q_a, Q_p, \Delta, Actions, Labels_p, Init)$*, where*

- Q_a *is the set of action nodes and* Q_p *is the set of probabilistic nodes. We define the set of all nodes as* $Q := Q_a \cup Q_p$*.*
- Actions *is the finite set of possible system actions, whereas* Labels$_p$ *denotes the finite set of probabilistic labels. Actions and* $Labels_p$ *shall be disjunct.*
- $\Delta_a \subseteq (Q_a \times Actions \times Q)$ *is the action transition relation, whereas* $\Delta_p \subseteq (Q_p \times (0, 1] \times Labels_p \times Q)$ *denotes the probabilistic transition relation with the second entry of each tuple being the assigned probability. We define* $\Delta := \Delta_a \cup \Delta_p$*.*
- Init $\subseteq Q$ *is the set of initial nodes.*

Further on, we need some constraints on those systems:

Definition 6 (Constraints on MDPs)**.**

1. *We prohibit that two probabilistic transitions coming from the same node, entering the same node, and having the same label, can have different probabilities.*
2. *We additionally prohibit nondeterminism in the action transitions, thus every action shall only be selectable at most once from every action node.*
3. *The sum of all probabilities of transitions going out of a probabilistic node has to be 1.*
4. *Every action node has at least one successor node.*

To visualize this definition, an example MDP is presented in Figure 3.

Since we want to reason about MDPs later on, we need to define some terminology regarding runs. A *run* on an MDP $\mathfrak{M}$ is uniquely determined by the sequence of nodes and labels. Thus, we define a run as a word $r \in (Q \cdot Labels)^\omega$ where for every prefix $q_i l_i q_{i+1}$ holds that there exists a transition from q_i to q_{i+1} labeled with l_i. It does not matter whether this transition is probabilistic or not. If the first letter of r is an initial node, then r is called an *initial run*. The set of all runs on $\mathfrak{M}$ is written as $Runs(\mathfrak{M})$, whereas the set of all runs on $\mathfrak{M}$ starting in q is written as $Runs(\mathfrak{M}, q)$.

A *path* of a given run r is a (finite) prefix of r. The set of all paths of a given MDP $\mathfrak{M}$ is called $Paths(\mathfrak{M})$. Again, the set of paths starting in q is denoted by $Paths(\mathfrak{M}, q)$.

The *trace* t_r of a run r is defined as the sequence of labels induced by the run. This means we take r and remove every occurrence of nodes in it, thus the remaining word becomes the trace t_r. We

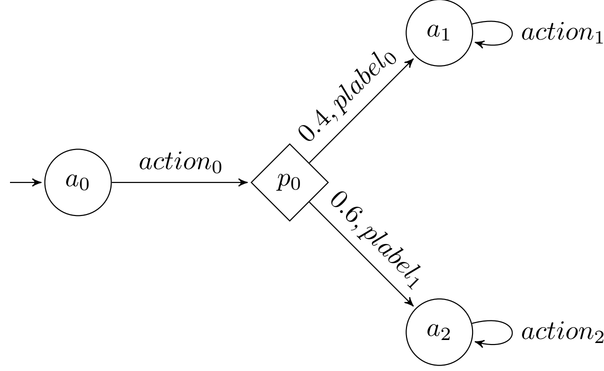

Figure 3: An example MDP $\mathfrak{M}_0$ where $\mathfrak{M}_0 = (Q_a, Q_p, \Delta, Actions, Labels_p, Init)$ with the nodes $Q_a = \{a_0, a_1, a_2\}$ and $Q_p = \{p_0\}$, the system actions $Actions = \{action_0, action_1\}$ and probabilistic labels $Labels_p = \{plabel_0\}$. The set of initial nodes consists of one node a_0.

can further restrict that trace to a particular alphabet Σ by removing every occurrence of labels that are not in Σ. We denote this *restricted trace* by $t_r[\Sigma]$.

To understand those notions we take a look at our example MDP in Figure 3. The runs of this automaton have the form $R_1 = a_0 \cdot action_0 \cdot p_0 \cdot plabel_0 \cdot (a_1 \cdot action_1)^\omega$ and $R_2 = a_0 \cdot action_0 \cdot p_0 \cdot plabel_1 \cdot (a_2 \cdot action_2)^\omega$, where some run has a chance of 0.4 to be in the first omega-language and 0.6 to be in the latter. $r = a_0 \cdot action_0 \cdot p_0 \cdot (a_1 \cdot action_1)^5$ would be one specific path of a run $r \in R_1$. The trace t_r of a run $r \in R_1$ has the form $action_0 \cdot plabel_0 \cdot (action_1)^\omega$. We may want to restrict it to contain only the actions and no probabilistic labels, thus we get $t_r[Actions] = action_0 \cdot (action_1)^\omega$. Notice that traces can be finite and infinite.

Since we now defined the semantics of MDPs, we need to translate a given probabilistic program into such an MDP. By doing this, we can argue about concrete runs of a program just by examining the runs of the associated MDP. Similar to [5], we will do so by setting the nodes as the possible program configurations. With suitable actions we are then able to move from one program configuration to another. This is best explained using an example: Starting from the program configuration `{k → 0, i → 0}` the MDP should allow to proceed to the configuration `{k → 1, i → 0}` if the program code states `k = k + 1`. As stated earlier, we are going to employ the probabilistic transitions for the `coin(p)` statements of our program. Since we also allow non-deterministic programs we need to incorporate those into the MDP as well. This is where we need to differentiate between action nodes and probabilistic nodes. For our construction, probabilistic transitions are used to change states deterministically. Since not every deterministic statement incorporates the `coin`-method, we set the probability to 1 for non-probabilistic, deterministic configuration transitions. We will use action nodes only to indicate a non-deterministic program part (namely anything involving `nondet()`). Here we allow our configuration to proceed to two new configurations, one for every possible resulting program state.

Definition 7 (Semantics of probabilistic programs). *Let $P = (Locations, Init^P, \hookrightarrow, Labels, \bot, \top)$ be a probabilistic program as defined in Definition 2.*

We define an MDP $\mathfrak{M}_P = (Q_a, Q_p, \Delta, Actions, Labels_p, Init)$ where

- *$Q_a = Locations_n \times Conf$, thus the action nodes are tupels containing a non-deterministic location and a program configuration.*
- *$Q_p = (Locations_d \times Conf) \cup \{\top\}$. The probabilistic nodes assign a program configuration to a deterministic location.*
- *$Actions = \{n_0, n_1\}$ denotes the actions whose function is to indicate which non-deterministic path in the exponential run-tree we chose on a specific run. n_0 indicates that `nondet()` was 0 in this run while n_1 does the same for 1.*
- *$Labels_p = \{c_0, c_1, -\}$ analogously defines the probabilistic labels. For every deterministic action there has to be a corresponding label: We use c_0 and c_1 if `coin(p)` yielded a 0 respectively*

1 and $-$ to indicate that no probabilistic expression was used.

- *$Init = Init^P$ is the set of initial program configurations. Note that it may be infinite.*

For convenience, we assume from now on that P is given in a normal form where expressions are not mixed or nested in types, i.e. we only allow – for $x \in \{0,1\}$ – `coin(p) == x`, `nondet() == x` or a non-deterministic non-probabilistic expression e. One can easily see that this normal form does not restrict the expressiveness of our programs, since every statement of the form `if(e ∘ coin(p)) {...}` for an boolean operator $\circ$ and an expression `e` can be represented by `x = coin(p); if(e ∘ x) {...}`, and accordingly for nondeterministic expressions.

Furthermore, we will need to define the transition relation between the nodes, which is based on the definition given in [6].

Definition 8 (Transition relation of $\mathfrak{M}_P$)**.** *The transition relation Δ is defined as following: For every node $v = (n, \sigma)$ of $\mathfrak{M}_P$ and every edge (n, n') of P*

- *if $Labels(n, n') =$ `x = e` for a deterministic, non-probabilistic expression e then add $(v, 1, -, (n', \sigma[x = e(\sigma)]))$ to Δ, where $\sigma[\alpha]$ denotes the updated configuration where α was applied. $e(\sigma)$ defines that expression e is evaluated under the current configuration σ.*
- *if $Labels(n, n') =$ `x = coin(p)` then add $(v, p - 1, c_0, (n', \sigma[x = 0]))$ and $(v, p, c_1, (n', \sigma[x = 1]))$ to Δ.*
- *if $Labels(n, n') =$ `x = nondet()` then add $(v, n_0, (n', \sigma[x = 0]))$ and $(v, n_1, (n', \sigma[x = 1]))$ to Δ.*
- *if $Labels(n, n') =$ `if(e)` for an expression e then*
  - *if e = `coin(p) == x` for $x \in \{0,1\}$ then add $(v, p, c_x, (n', \sigma))$ to Δ.*
  - *if e = `nondet() == x` for $x \in \{0,1\}$ then add $(v, n_x, (n', \sigma))$ to Δ.*
  - *if e does not contain both above stated functions then we just add a ordinary condition transition to the MDP, but only if the current program configuration σ satisfies e: $(v, 1, -, (n', \sigma))$.*
- *if $n = \top$ then add $(v, 1, -, \top)$ to Δ.*

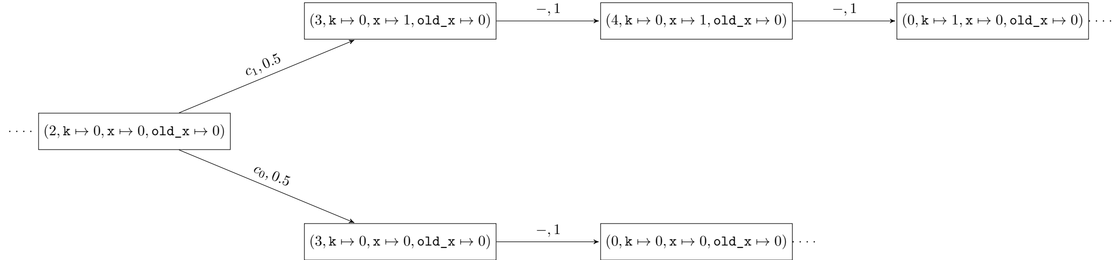

Figure 4: A fragment of the MDP belonging to Example Program 1. The probabilistic nodes are represented as rectangles. Note that there are no action nodes present due to the lack of non-deterministic commands in program 1.

This rather long-described construction can be examined by the use of Example 4. For the non-deterministic, non-probabilistic commands the "$-, 1$" labeling is used which indicates that no coin is thrown ($-$) and that the probability of the transition is 1. Those transitions are present for the initial program command `k = 0` and for the decrements and increments of k depending on the coin toss outcome. Since we incorporated `coin(0.5)` into the example, we need to split up the program path in two possible resulting paths, indicated by c_0 and c_1. Note that the probabilities add up to 1 as constrained before.

For obvious reasons, only a partial construction and not the full MDP is shown. Since the state space contains every possible variable valuation, it is theoretically infinite. But even if we restrict

it to 64-bit integers, for 2 program variables we need to construct at maximum $(2^{64})^2$ states. One may notice that this problem is practically unsolvable. This is among other things the reason why termination checkers and other verification tools won't construct the MDP for their inputs. Likewise, our later presented algorithm will not construct an MDP explicitly. But since they are important to show termination properties of probabilistic programs, they are nevertheless needed.

2.1.3 Types of probabilistic programs

In this paper we need to distinguish between two program classes: Finite and weakly finite programs.

Definition 9 (Finite and weakly finite programs)**.** *The class of finite programs contains all programs P for which it holds that $\mathfrak{M}_P$ has only a finite number of nodes reachable from every initial node. For weakly finite programs we additionally allow an infinite number of initial nodes.*

As may be imagined, the termination proofs of weakly finite programs turn out to be more complex than for finite programs, since we need to respect every possible initial program configuration. Weakly finite programs include the class of parameterized programs. Figure 5 shows such a weakly finite program. Just as Example 1 it terminates almost surely, but termination is not as obvious: For an arbitrary large parameter `N` the coin tosses need to yield 1 `N` times in a row – But since we need to fix the input before we even start the program, we can be sure that such a number of coin toss outcomes will be eventually reached.

```
1 k = 0;
2 x = 0;
3 while(k < N) {
4     old_x = x;
5     x = coin(0.5)
6     if(x != old_x)
7         k = k + 1;
8 }
```

Figure 5: An example weakly finite program which takes `N` as an input parameter. It appears quite similar to program 1, but yet a termination proof is not as obvious to find.

2.2 Patterns

Since we now defined probabilistic programs in syntax and semantics, we will recall to the introduction where it was stated that *Patterns* will be used to show termination of a given probabilistic programs. A pattern is, broadly speaking, a family of sequences induced by coin-toss outcomes of concrete program runs.

Definition 10 (Patterns)**.** *A pattern Φ is a subset of C^ω where C is the set of possible coin toss outcomes. In this paper it will be fixed to $C = \{0, 1\}$. We will denote Φ as the following expression to indicate its structure: $\Phi = C^*w_1C^*w_2C^*w_3C^* \dots$ with $w_i \in C^*$.*

Thus we define patterns as a *language* over C by using a (possibly infinite) concatenation of regular expressions over finite languages. Note that the structural representation $C^*w_1C^*w_2C^*w_3C^* \dots$ does not limit the expressiveness of patterns. The words w_i represent the essential part of the coin tosses in which we are particularly interested in. The C^* parts of the pattern denote finite but irrelevant parts for the termination proof. It is of importance that after getting w_i as a coin toss outcome sequence, we will eventually see w_{i+1} later.

Definition 11 (The universal pattern.)**.** *Since C is a finite set, there exists an enumeration $x_1, x_2, \dots$ of C^*. Using this enumeration, we denote the* universal pattern *by $C^*x_1C^*x_2C^* \dots$.*

The universal pattern guarantees us to reach every possible finite coin toss sequence somewhen in a program execution. This will prove helpful later on. An example for a universal pattern is the pattern $\Phi_u = C^*0C^*1C^*00C^*01C^*10C^*11C^*000 \dots$.

Definition 12 (Pattern-conforming runs). *Let P be a probabilistic program and Φ a pattern. A run r of P is* Φ-conforming *if the trace $t_r[C]$ is a prefix of some word in Φ, i.e. the coin toss outcomes of the run match the structure defined by the pattern. We denote the set of Φ-conforming runs as $Runs(\mathfrak{M}, \Phi)$.*

Definition 13 (Terminating pattern). *A pattern Φ is* terminating *if every Φ-conforming run eventually reaches the final state $\top$, i.e. the run terminates.*

Let us examine the notion of patterns by the use of Example Program 1. As already observed, the program terminates if we see ten outcome alternations in the coin tosses. We are now able to formalize this intuitive thinking: We define our terminating pattern $\Phi_t = (C^*01)^{10}C^\omega$, so $w_i = 01$ for every $i \in \{1, \dots, 10\}$.

The exemplary trace $t_r = 100110(01)^{10}$ of a (terminating) run r would be Φ_t-conforming, since t_r is a prefix of a word in Φ_t (e.g. $100110(01)^{10}0^\omega$).

We can also define non-terminating patterns such as $\Phi_n = C^*0C^*00C^*000C^* \dots$. Observe that we can not represent Φ_n as a regular expression over ω-words. One should also note that some Φ_n-conforming runs may terminate, e.g. the run r with its trace $t_r = (01)^{10}0^\omega$. But since not *all* runs in Φ_n terminate (e.g. r' with $t_{r'} = 0^\omega$) the pattern is not terminating itself. Thus, the important property of terminating patterns is the guarantee of every conforming run being a terminating one.

3 Almost-sure termination

In this section, we define the meaning of "termination with probability one" and also prove the correctness of the pattern approach. For this section, we fix a probabilistic program P and its associated MDP $\mathfrak{M}_P$.

Definition 14 (Almost-sure termination). *A probabilistic program P is* terminating almost surely *if the probability of all terminating runs of the associated MDP $\mathfrak{M}_P$ starting in n_i is 1 for every initial node n_i. It is required to hold for every discrete probability measure to which* `coin(p)` *may evaluates (e.g. the discrete uniform distribution).*

Using patterns, we have now defined a construct that allows us reason about termination of probabilistic programs. Looking back to the introduction, we stated the following claim which is still to prove:

Claim 1. *Let R be a set of runs on a probabilistic program P. If the probability of R is 1 and every run in R is terminating, then P terminates almost surely.*

We will break the claim into two independent pieces:

Theorem 1. *Let Φ be a not further constrained pattern of P. Then the probability of the set of Φ-conforming runs is 1.*

Theorem 2. *Let Φ be a terminating pattern of P. Then it follows that P is almost sure terminating.*

Further on, a proof of Theorem 1 will be outlined. For a detailed version of the proof, see the appendix of [5].

Sketch of a proof of Theorem 1. Let $\Phi = C^*w_1C^*w_2C^*w_3C^* \dots$ be a pattern on $\mathfrak{M}_P$. We want to show that $P(Runs(\mathfrak{M}, \Phi)) = 1$, where $P(A)$ denotes the probability of the event A. A formal definition of discrete and countable probability spaces and its measures can be found on Page 167 and following in [1].

Since $Runs(\mathfrak{M}, C^\omega \setminus \Phi) = Runs(\mathfrak{M}) \setminus Runs(\mathfrak{M}, \Phi)$ contains all runs that are not induced by the pattern Φ, we can examine whether this set of run has the probability 0. In stochastic notions this would be called a *null set*. If this is determined, it follows that the probability of the set of runs induced by Φ is 1.

It is easy to see that

$$Runs(\mathfrak{M}, C^* w_1 w_2 \dots w_n C^\omega) \subseteq Runs(\mathfrak{M}, \Phi) \subseteq Runs(\mathfrak{M}) \ (*)$$

holds for every $n \in \mathbb{N}$. For example, the set of runs which are induced by $\Phi = C^* 1 C^* 0 C^\omega$ of Program 1 (namely every run which eventually has an ensuing 1 after a 0 coin toss) is a subset of all runs. The set of runs induced by $C^* 10 C^\omega$ are again defined as all runs which have the coin-toss sequence 10 somewhen present. This set is obviously a subset of $Runs(\mathfrak{M}, \Phi)$.

Thus, it is now sufficient to show that $P(Runs(\mathfrak{M}, C^* w_1 w_2 \dots w_n C^\omega)) = 1$.

Or, if use the above mentioned approach, it is sufficient to show that

$$P(Runs(\mathfrak{M}) \setminus Runs(\mathfrak{M}, C^* w_1 w_2 \dots w_n C^\omega)) = 0$$

holds.

We will do so by estimating an upper bound for $P(S)$ where S is the set of non-$C^* w_1 \dots w_n C^\omega$-conforming runs, thus $S = Runs(\mathfrak{M}) \setminus Runs(\mathfrak{M}, C^* w_1 w_2 \dots w_n C^\omega)$. We do so by examining the probability of the set of runs which do not contain $w = w_1 w_2 \dots w_n$, but visit more than $|w|$ many probabilistic nodes. Intuitively, we need to visit more than $|w|$ many nodes because otherwise it would not be possible to achieve n coin tosses in such a run and ultimately, since they will never by induced by our pattern, those runs are of no interest to our observations – We need to exclude them from our analysis.

We will denote this set by $B(j) = V(j \cdot |w|) \cap (Runs(\mathfrak{M}) \setminus S)$ where $V(j)$ is the set of runs which visit a probabilistic node at least j times.

We are then able to show that

$$P(Runs(\mathfrak{M}) \setminus S) = \lim_{j \to \infty} P(B(j))$$

and then estimate following upper bound

$$\lim_{j \to \infty} P(B(j)) \leq \lim_{j \to \infty} (1 - p_{min}^n)^j$$

where p_{min} denotes the minimal probability that appears in $\mathfrak{M}$. Since $p_{min} > 0$ it follows that

$$\lim_{j \to \infty} (1 - p_{min}^n)^j = 0.$$

Thus, we conclude that

$$P(S) = 0$$

and finally, since (*) holds, it follows that

$$P(Runs(\mathfrak{M}, \Phi)) = 1.$$

□

Since we are now aware that $P(Runs(\mathfrak{M}, \Phi)) = 1$ for every pattern Φ of P, it remains to show the second part of our essential Claim 1.

Proof of Theorem 2. Let Φ be a terminating pattern and $R_t = Runs(\mathfrak{M}, \Phi)$. We denote the set of all terminating runs of $\mathfrak{M}$ by $Runs_t(\mathfrak{M})$.

It follows by Theorem 1 that $P(R_t) = 1$. Because R_t is a terminating pattern, every run $r \in R_t$ is terminating. Thus, $R_t \subseteq Runs_t$. Since $P(R_t) = 1$ holds, $P(Runs_t) = 1$ is a direct consequence of the subset relation.

By definition of almost sure termination it follows that P is terminating almost surely. □

4 The algorithm

We have now gathered all essential constructs and theorems together to present the termination prover devised by [5]. We will need to distinguish between finite and weakly finite programs, since finite programs – due to a fixed initial configuration – allow for an easier algorithm design.

In general, we will divide our algorithm into two pieces:

1. *The pattern constructor*
2. *The pattern checker*

While the pattern constructor builds our hopefully-to-find terminating pattern, the pattern checker verifies whether our just constructed pattern is a terminating one.

Furthermore, we will use an iterative, refinement-based approach on the pattern constructor, which basically refines the pattern in every iteration and verifies subsequently if that pattern is terminating. In the loop body of our algorithm, we try to scrape up information we have: Φ is of a form we know *and* Φ is not terminating. Thus, we will need to reconstruct Φ in such a way that the chances increase that Φ will become terminating.

4.1 The pattern constructor

In this subsection we will just assume that we have already built such a pattern checker, which returns "True" if a given pattern Φ terminates on a given program P, and a counterexample otherwise. We will call those counterexamples *lassos*. A lasso is a trace of a non-terminating run r of P where $t_r \in \Phi$. We will denote the periodically repeating part of this run as the *loop*.

4.1.1 Finite programs

As proposed in [5], every finite program has a *simple terminating pattern*, which is defined as $\Phi_s = (C^* w)^\omega$ for some word $w \in C^*$. We will utilize this statement and let our algorithm construct a simple terminating pattern, i.e. instead of the whole pattern Φ we only need to refine our word w in every iteration.

Data: A probabilistic program P and a baseword $s_0 \in C^*$.
Result: *True* if P terminates almost surely, *false* otherwise.
$i := 0$
while $(C^* s_i)^\omega$ *is not a terminating pattern* **do**
  l_i := lasso, taken from the termination checker.
  u_i := loop of l_i.
  if $u_i = \epsilon$ **then**
    return *false*
  else
    s_{i+1} := shortest word that has s_0 as prefix and is not an infix of any u_k^ω for $k \in \{1, \ldots, i\}$.
  end
  $i := i + 1$.
end
return *true*

Figure 6: The pattern-constructor for finite programs. By default the baseword s_0 is set to ϵ.

The pseudo code of the algorithm is shown in Figure 6. The algorithm takes a probabilistic program P and an arbitrary baseword s_0, which by default is set to ϵ. As previously mentioned, the constructed pattern is iteratively refined.

In every loop iteration, the termination checker is executed on P with the beforehand constructed pattern. The pattern checker then returns a counterexample if the pattern is non-terminating.

This counterexample is subsequently used to refine our word s_i: We construct s_i such that it does not contain *any* coin-toss sequence induced by the previously detected loops. Additionally, s_i has s_0 as a prefix, thus we prepend the baseword to every repeating pattern element s_i. By default, where $s_0 = \epsilon$, s_i remains unchanged. We also demand s_i to be the shortest word for which the desired properties hold. This prevents that we may "skip" infinite loops of P in our loop detection.

Thus, in each iteration the algorithm constructs the word s_i such that it does not evoke the previously detected loops in its induced program runs. This guarantees us that $C^* s_i C^\omega$ will eventually transform into the wanted simple terminating pattern if P terminates almost surely.

In the case that the pattern checker did not find a run with a looping part (thus, $u_i = \epsilon$), we know that P is not almost sure terminating, since P will loop on the run l_i with only finite many coin tosses. For example, this is the case if a program with a non-terminating loop with no probabilistic functions is given.

The functioning is best examined by the use of an example. Again, we consider our finite Program 1. The algorithm then constructs the following patterns, starting with $\Phi = (C^*\epsilon)^\omega$, in every iteration step:

1. The pattern checker yields 0^ω as a counterexample, thus we construct $(C^*1)^\omega$, since 1 is the shortest word that is not an infix of 0^ω.
2. The pattern checker yields 1^ω as a counterexample, thus we construct $(C^*01)^\omega$, since 01 is the shortest word that is neither an infix of 0^ω nor 1^ω.
3. The pattern checker then determines that $(C^*01)^\omega$ is a terminating pattern – The algorithm returns "True".

4.1.2 Weakly finite programs

For weakly finite programs, the pattern construction needs to be a bit more sophisticated, since we allow for infinitely many initial nodes. The algorithm will require human interaction. The main idea is to construct a program P' from the given program P where P' fixes one of those initial nodes as its only initial node. This results in P' being finite and ultimately allows us to run our previously constructed algorithm for finite programs on it. Since the set of initial nodes of P is countable, the presented algorithm just iterates through every possible P'.

Formally, the algorithm presented in Figure 7 fixes an enumeration $i_1, i_2, \dots$ of Init^P. For every element i_k of this enumeration the previously mentioned probabilistic program P_{i_k} is constructed, where $P_{i_k} = (\mathit{Locations}^P, \{i_k\}, \hookrightarrow^P, \mathit{Labels}^P, \perp^P, \top^P)$, thus equal to P but fixing i_k as the only initial node.

After running the termination checker for finite programs on P_{i_k} using the baseword $w_{i_{k-1}}$, we are given a simple terminating pattern of the form $(C^* w_{i_k})C^\omega$. Note that every previously constructed w_{i_j} is a prefix of w_{i_k}. For every initial node we will receive such a pattern. Since $(C^* w_{i_k})^\omega$ is terminating for the initial node i_k, we also know that, for every $n \in \mathbb{N}$ the pattern $\Phi_n = C^* w_{i_1} C^* w_{i_2} C^* \dots w_{i_n} C^\omega$ is terminating for the program $P_n = (\mathit{Locations}^P, \{i_1, \dots, i_n\}, \hookrightarrow^P, \mathit{Labels}^P, \perp^P, \top^P)$, thus the program derived from P where the initial nodes are fixed to $i_1, \dots, i_n$.

We then keep on computing the pattern Φ_n. If Init^P is infinite, the algorithm will never halt. But practical observations have shown that it is often sufficient to only compute the first few patterns and then let a human extrapolate this pattern, albeit there is no theoretical evidence that those patterns are always easily predictable. However, since most programs are written by humans, they are likely to contain some predictable loop structures which often induce straightforward patterns. As an example, the algorithm may construct $\Phi_1 = C^*0C^\omega$, $\Phi_2 = C^*0C^*00C^\omega$ and $\Phi_3 = C^*0C^*00C^*000C^\omega$. Using human intuition, we can extrapolate the k-th word as $w_{i_k} = 0^i$ from this information.

After the human has found such a possible terminating pattern, we will pass it to our assumed to be constructed pattern checker. It will then verify whether the intuitive approach was correct or not. Note that we use the pattern checker for weakly finite programs in this step, as opposed to before where we used the pattern checker for finite programs on the pattern Φ_n.

Data: A weakly finite probabilistic program P.
Result: *True* and the terminating pattern if P terminates almost surely, *false* otherwise.
Fix an enumeration $i_1, i_2, \ldots$ of $Init^P$.
$k := 0$
while *true* **do**
  Construct P_{i_k}.
  if P_{i_k} *is almost-sure terminating* **then**
    Φ_{i_k} := simple terminating pattern $C^* w_{i_k} C^\omega$ of P_{i_k} using $w_{i_{k-1}}$ as a baseword.
    $\Phi_k := C^* w_{i_1} C^* w_{i_2} \ldots C^* w_{i_k} C^\omega$.
    if *human is able to extrapolate a sequence* $(w_{i_n})_{n \in \mathbb{N}}$ *when given* Φ_k **then**
      if $\Phi = C^* w_{i_1} C^* w_{i_2} \ldots$ *is a terminating pattern* **then**
        return *true and* Φ.
      end
    end
    $k := k + 1$
  else
    return *false*
  end
end
return *true*

Figure 7: The pattern constructor for weakly finite programs. In its iteration steps, it calls the in Figure 6 presented algorithm for finite programs.

4.2 The pattern checker

It remains to construct a pattern checker. A pattern checker takes a pattern Φ and a probabilistic program P to verify the termination property for. Again, we need to distinguish between finite and weakly finite programs, since the latter program class enforces a more elaborated algorithm design.

4.2.1 Finite programs

For finite programs, we can easily exploit the strength of already built model-checkers, in this particular outline we will use SPIN [8], as mentioned in the introduction.

The pattern checker for finite programs takes a simple pattern of the form $\Phi = (C^* w)^\omega$ and the program P. If Φ is a terminating pattern it shall return "True", otherwise it shall return a lasso l as a counterexample, which is a run of Φ that is non-terminating.

Since SPIN is not able to work with probabilistic programs, we need to eliminate the probabilistic parts of P. We do so by using non-determinism to emulate the different probabilistic outcomes. We establish two unused variables named `c` $\in \{0, 1, 2\}$ and `termination` $\in \{0, 1\}$. If `termination` equals 1, the program will halt in the next step. Otherwise, `termination` will always be set to 0. `c` indicates the different probabilistic coin-toss outcomes: If `c` was set to 0, the `coin(p)`-function yielded 0, and equally for 1. Directly after every probabilistic expression, `c` is set to 2, so both program paths won't differ from this point on.

Formally, the transformation from probabilistic to non-probabilistic non-deterministic programs is shown in Figure 8. For this step, we will assume that P is in a normal-form where it does not contain `if(coin(p))`-conditions. Those are beforehand transformed into the code snippet `x = coin(p); if(x) {...}`, where x is an unused variable.

Further on, we will incorporate the `termination` variable such that it is set to 0 at the very beginning of the program and to 1 at the very end.

Now, we are able to verify properties of the program using a model checker for non-deterministic non-probabilistic programs, such as SPIN. Since we need to pass an LTL-formula to spin, we need to express the non-termination property in such a formula: In every execution of P, `termination`

```
x = coin(p);
```

$\rightsquigarrow$

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

Figure 8: The transformation from probabilistic to non-probabilistic programs. In this process, the program will unavoidable become non-deterministic, as indicated by the `nondet()`-function. Since `coin(p)` is defined by a discrete probability measure, the method is only applicable to programs which work on such a discrete stochastic model.

shall always be 0, and additionally the run shall only incorporate finitely many `coin(p)`-calls, thus eventually stop visiting the probabilistic locations of P. The formula $\varphi = \neg termination \wedge FG(c \notin \{0,1\})$ expresses those circumstances.

We will pass the now modified program P along with the property φ to SPIN, which then returns True if there exists a run of P which is non-terminating and visits only finitely many probabilistic locations. The pattern checker will return this lasso. Otherwise SPIN reports False, thus no run of P exists for which φ holds. This means that all non-terminating runs of P involve infinitely many probabilistic expressions.

Up to now, we did not incorporate the pattern Φ. We will do so by constructing a deterministic Büchi-automaton (DBA) $\mathfrak{A}$ with $L(\mathfrak{A}) = Runs(\mathfrak{M}, \Phi)$. Thus, $\mathfrak{A}$ represents all runs induced by Φ. Since we eliminated probabilistic expressions, $\mathfrak{A}$ represents the Φ-conforming runs via specifying the assignments to `c`.

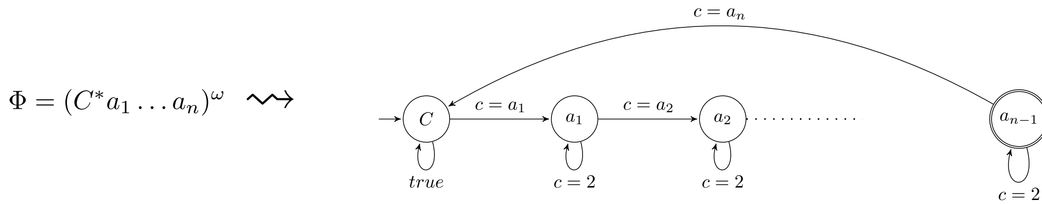

Figure 9: The transformation from a simple terminating pattern $\Phi = (C^* a_1 \dots a_n)^\omega$ into a Büchi-automaton $\mathfrak{A}$ with $L(\mathfrak{A}) = Runs(\mathfrak{M}_P, \Phi)$. $a_1, \dots, a_n$ are elements of the given alphabet C. *true* indicates that every command (except the labels of the outgoing transitions) can be read at this point.

We can now pass the DBA $\mathfrak{A}$ along with P to SPIN, which then verifies whether P has a run that is accepted by $\mathfrak{A}$. If this is the case, we can simply return the now determined lasso. Otherwise, the algorithm returns that Φ is a terminating pattern.

4.2.2 Weakly finite programs

For weakly finite programs the algorithm design again needs to be a bit more sophisticated since we need to check the termination property for every initial node. Also, not every weakly finite program has a simple terminating pattern. In fact, their terminating patterns are may not even NBA-recognizable, since we allow for general $\omega-$languages here.

Again, we employ a transformation from probabilistic to non-deterministic programs to pass the result into a model-checker. We will use ARMC for this, since this tool is able to handle infinite program state sets. For this, we denote `nondet(S)` as the function which nondeterministically returns a number from the given countable set `S`. We transform P into the non-deterministic program P^Φ which depends on the to be evaluated pattern Φ.

The actual transformation is shown in Figure 10.

```
x = coin(p);
```

$\rightsquigarrow$

```
// Initialization at the program start
counter  = nondet(ℕ);
next     = 1;
position = 1;
[...]
// Actual transformed part
x = nondet();
if (counter <= 0) {
    if (position > length(w[next])) {
        counter  = nondet(ℕ);
        position = 1;
        next     = next + 1;
    } else {
        x        = w[next][position];
        position = position + 1;
    }
} else {
    counter := counter - 1;
}
```

Figure 10: The transformation from probabilistic to non-probabilistic weakly finite programs, which depends on a given pattern Φ. With `w[i][k]` we denote the k-th element of the word w_i of the pattern $\Phi = C^* w_1 C^* w_2 \dots$. `counter`, `next`, `position` are newly introduced variables.

We use `counter` to guess the length of the irrelevant part of the program run, namely those parts of the trace that are not part of any w_i. With this transformation we achieve that all program runs conform to the given pattern Φ, because we "cut off" the C^* parts of the run. We assign to `x`, in a non-deterministically fashion, the content of the every w_i that appears in the pattern – And jump, via the `counter` variable, over the irrelevant parts.

Since every run of P^Φ is Φ-conforming, and every run induced by Φ is a run of P^Φ, we can then pass P^Φ to our model-checker. Since ARMC is able to prove termination without any further input, we will just query ARMC for this purpose. If ARMC is able to prove termination of P^Φ we know that every run of P^Φ is a terminating one. Since the runs of P^Φ are exactly the runs of P induced by Φ, we can report that Φ is a terminating pattern. If ARMC returns an infinite run of P^Φ we are able to report that Φ is not a terminating pattern of P, and we return the lasso that ARMC yielded. Note that ARMC may gives up on P^Φ, due to the undecidability of the halting problem. In this case, the presented algorithm also surrenders and returns "Maybe".

5 Conclusion and applications

In this paper, the in [5] devised algorithm was presented. This algorithm is able to prove termination of finite programs and, employing human ingenuity, tries to prove termination of weakly finite programs. After transforming probabilistic programs into non-probabilistic non-deterministic programs, we were able to use SPIN and ARMC to verify the termination of our iteratively refined pattern. For weakly-finite programs, where ARMC was employed, the algorithm took way longer to compute compared to SPIN on finite programs [5]. This is not surprising, since ARMC needs to verify the termination property of a program with infinitely many initial nodes. Experiments of [5] also yielded that the runtime of the employed non-probabilistic model-checkers SPIN and ARMC accounts for the main part of the computation. Thus, the best way to improve the algorithm's runtime is to enhance the employed tools. The authors of [5] were actually able to modify ARMC in such a way that a loop invariant would be found faster.

Termination provers of probabilistic programs are needed in a great range of applications. In randomized network protocols probabilistic expressions are for instance used to retrieve sensor data in a probabilistic manner, as presented in [7]. Thus, the termination of the retrieving process is of utter importance: If the process never halts, no sensor data is returned and the sensor itself stays useless in that time. Therefore, we need to at least attempt a termination proof – And, when using the presented algorithm, with no or little human interaction involved.

References

[1] R.B. Ash and C. Doléans-Dade. *Probability and Measure Theory.* Harcourt/Academic Press, 2000.

[2] C. Baier and I. Katoen. *Principles of Model Checking.* 2008.

[3] D. Bert. *ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings.* Lecture Notes in Computer Science. Springer, 2003.

[4] M. Davis. *Computability & Unsolvability.* Dover Books on Computer Science Series. Dover, 1958.

[5] Javier Esparza, Andreas Gaiser, and Stefan Kiefer. Proving termination of probabilistic programs using patterns. *CoRR*, abs/1204.2932, 2012.

[6] Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Perform. Eval.*, 73:110–132, 2014.

[7] M. Hefeeda and H. Ahmadi. A probabilistic coverage protocol for wireless sensor networks. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on.*

[8] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

[9] A. Podelski and A. Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *LNCS 4354*, 2007.