Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT*
(Short Paper)

Nils Lommen(✉), Éléanore Meyer, and Jürgen Giesl(✉)

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
{lommen,eleanore.meyer,giesl}@cs.rwth-aachen.de

Abstract. Recently, we showed how to use control-flow refinement (CFR) to improve automatic complexity analysis of integer programs. While up to now CFR was limited to classical programs, in this paper we extend CFR to *probabilistic* programs and show its soundness for complexity analysis. To demonstrate its benefits, we implemented our new CFR technique in our complexity analysis tool KoAT.

1 Introduction

There exist numerous tools for complexity analysis of (non-probabilistic) programs, e.g., [2–6, 10, 11, 15, 16, 18, 19, 24, 25, 28, 30, 32]. Our tool KoAT infers upper runtime and size bounds for (non-probabilistic) integer programs in a modular way by analyzing subprograms separately and lifting the obtained results to global bounds on the whole program [10]. Recently, we developed several improvements of KoAT [18, 24, 25] and showed that incorporating control-flow refinement (CFR) [13, 14] increases the power of automated complexity analysis significantly [18].

There are also several approaches for complexity analysis of *probabilistic* programs, e.g., [1, 7, 9, 21–23, 27, 29, 31, 34]. In particular, we also adapted KoAT's approach for runtime and size bounds, and introduced a modular framework for automated complexity analysis of probabilistic integer programs in [27]. However, the improvements of KoAT from [18, 24, 25] had not yet been adapted to the probabilistic setting. In particular, we are not aware of any existing technique to combine CFR with complexity analysis of probabilistic programs.

Thus, in this paper, we develop a novel CFR technique for probabilistic programs which could be used as a black box by every complexity analysis tool. Moreover, to reduce the overhead by CFR, we integrated CFR natively into KoAT by calling it on-demand in a modular way. Our experiments show that CFR increases the power of KoAT for complexity analysis of probabilistic programs substantially.

The idea of CFR is to gain information on the values of program variables and to sort out infeasible program paths. For example, consider the probabilistic **while**-loop (1). Here, we flip a (fair) coin and either set x to 0 or do nothing.

$$\textbf{while}\ \ x > 0\ \ \textbf{do}\ \ x \leftarrow 0\ \oplus_{1/2}\ \texttt{noop}\ \ \textbf{end} \tag{1}$$

* funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL

The update $x \leftarrow 0$ is in a loop. However, after setting x to 0, the loop cannot be executed again. To simplify its analysis, CFR "unrolls" the loop resulting in (2).

$$\begin{aligned} &\textbf{while } x > 0 \textbf{ do } \texttt{break} \oplus_{1/2} \texttt{noop} \textbf{ end} \\ &\textbf{if } x > 0 \textbf{ then } x \leftarrow 0 \textbf{ end} \end{aligned} \tag{2}$$

Here, x is updated in a separate, *non-probabilistic* **if**-statement and the loop does not change variables. Thus, we sorted out paths where $x \leftarrow 0$ was executed repeatedly. Now, techniques for probabilistic programs can be used for the **while**-loop. The rest of the program can be analyzed by techniques for non-probabilistic programs. In particular, this is important if (1) is part of a larger program.

We present necessary preliminaries in Sect. 2. In Sect. 3, we introduce our new control-flow refinement technique and show how to combine it with automated complexity analysis of probabilistic programs. We conclude in Sect. 4 by an experimental evaluation with our tool KoAT. We refer to [26] for further details on probabilistic programs and the soundness proof of our CFR technique.

2 Preliminaries

Let $\mathcal{V}$ be a set of variables. An *atom* is an inequation $p_1 < p_2$ for polynomials $p_1, p_2 \in \mathbb{Z}[\mathcal{V}]$, and the set of all atoms is denoted by $\mathcal{A}(\mathcal{V})$. A *constraint* is a (possibly empty) conjunction of atoms, and $\mathcal{C}(\mathcal{V})$ denotes the set of all constraints. In addition to "$<$", we also use "$\geq$", "$=$", etc., which can be simulated by constraints (e.g., $p_1 \geq p_2$ is equivalent to $p_2 < p_1 + 1$ for integers).

For *probabilistic integer programs (PIPs)*, as in [27] we use a formalism based on transitions, which also allows us to represent **while**-programs like (1) easily. A PIP is a tuple $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$ with a finite set of program variables $\mathcal{PV} \subseteq \mathcal{V}$, a finite set of locations $\mathcal{L}$, a fixed initial location $\ell_0 \in \mathcal{L}$, and a finite set of general transitions $\mathcal{GT}$. A *general transition* $g \in \mathcal{GT}$ is a finite set of transitions which share the same start location ℓ_g and the same guard φ_g. A *transition* is a 5-tuple $(\ell, \varphi, p, \eta, \ell')$ with a *start location* $\ell \in \mathcal{L}$, *target location* $\ell' \in \mathcal{L} \setminus \{\ell_0\}$, *guard* $\varphi \in \mathcal{C}(\mathcal{V})$, *probability* $p \in [0, 1]$, and *update* $\eta : \mathcal{PV} \to \mathbb{Z}[\mathcal{V}]$. The probabilities of all transitions in a general transition add up to 1. We always require that general transitions are pairwise disjoint and let $\mathcal{T} = \biguplus_{g \in \mathcal{GT}} g$ denote the set of all transitions. PIPs may have *non-deterministic branching*, i.e., the guards of several transitions can be satisfied. Moreover, we also allow *non-deterministic (temporary) variables* $\mathcal{V} \setminus \mathcal{PV}$. To simplify the presentation, we do not consider transitions with individual costs and updates which use probability distributions, but the approach can easily be extended accordingly. From now on, we fix a PIP $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$.

Example 1. The PIP in Fig. 1 has $\mathcal{PV} = \{x, y\}$, $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, and four general transitions $\{t_0\}$, $\{t_{1a}, t_{1b}\}$, $\{t_2\}$, $\{t_3\}$. The transition t_0 starts at the initial location ℓ_0 and sets x to a non-deterministic positive value $u \in \mathcal{V} \setminus \mathcal{PV}$, while y is unchanged. (In Fig. 1, we omitted unchanged updates like $\eta(y) = y$, the guard `true`, and the probability $p = 1$ to ease readability.) If the general transition is a

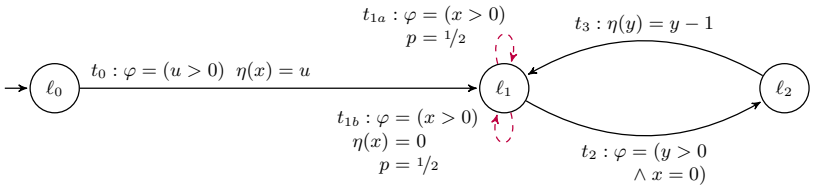

Fig. 1: A Probabilistic Integer Program

singleton, we often use transitions and general transitions interchangeably. Here, only t_{1a} and t_{1b} form a non-singleton general transition which corresponds to the program (1). We denoted such (probabilistic) transitions by dashed arrows in Fig. 1. We extended (1) by a loop of t_2 and t_3 which is only executed if $y > 0 \wedge x = 0$ (due to t_2's guard) and decreases y by 1 in each iteration (via t_3's update).

A *state* is a function $\sigma : \mathcal{V} \to \mathbb{Z}$, Σ denotes the set of all states, and a *configuration* is a pair of a location and a state. To extend finite sequences of configurations to infinite ones, we introduce a special location $\ell_\perp$ (indicating termination) and a special transition $t_\perp$ (and its general transition $g_\perp = \{t_\perp\}$) to reach the configurations of a run after termination. Let $\mathcal{L}_\perp = \mathcal{L} \uplus \{\ell_\perp\}$, $\mathcal{T}_\perp = \mathcal{T} \uplus \{t_\perp\}$, $\mathcal{GT}_\perp = \mathcal{GT} \uplus \{g_\perp\}$, and let $\mathsf{Conf} = (\mathcal{L}_\perp \times \Sigma)$ denote the set of all configurations. A *path* has the form $c_0 \to_{t_1} \cdots \to_{t_n} c_n$ for $c_0, \ldots, c_n \in \mathsf{Conf}$ and $t_1, \ldots, t_n \in \mathcal{T}_\perp$ for an $n \in \mathbb{N}$, and a *run* is an infinite path $c_0 \to_{t_1} c_1 \to_{t_2} \cdots$. Let Path and Run denote the sets of all paths and all runs, respectively.

We use Markovian schedulers $\mathfrak{S} : \mathsf{Conf} \to \mathcal{GT}_\perp \times \Sigma$ to resolve all non-determinism. For $c = (\ell, \sigma) \in \mathsf{Conf}$, a *scheduler* $\mathfrak{S}$ yields a pair $\mathfrak{S}(c) = (g, \tilde{\sigma})$ where g is the next general transition to be taken (with $\ell = \ell_g$) and $\tilde{\sigma}$ chooses values for the temporary variables such that $\tilde{\sigma} \models \varphi_g$ and $\sigma(v) = \tilde{\sigma}(v)$ for all $v \in \mathcal{PV}$. If $\mathcal{GT}$ contains no such g, we obtain $\mathfrak{S}(c) = (g_\perp, \sigma)$. For the formal definition of Markovian schedulers, we refer to [26].

For every $\mathfrak{S}$ and $\sigma_0 \in \Sigma$, we define a probability mass function $pr_{\mathfrak{S},\sigma_0}$. For all $c \in \mathsf{Conf}$, $pr_{\mathfrak{S},\sigma_0}(c)$ is the probability that a run with scheduler $\mathfrak{S}$ and the initial state σ_0 starts in c. So $pr_{\mathfrak{S},\sigma_0}(c) = 1$ if $c = (\ell_0, \sigma_0)$ and $pr_{\mathfrak{S},\sigma_0}(c) = 0$ otherwise.

For all $c, c' \in \mathsf{Conf}$ and $t \in \mathcal{T}_\perp$, let $pr_{\mathfrak{S}}(c \to_t c')$ be the probability that one goes from c to c' via the transition t when using the scheduler $\mathfrak{S}$ (see [26] for the formal definition of $pr_{\mathfrak{S}}$). Then for any path $f = (c_0 \to_{t_1} \cdots \to_{t_n} c_n) \in \mathsf{Path}$, let $pr_{\mathfrak{S},\sigma_0}(f) = pr_{\mathfrak{S},\sigma_0}(c_0) \cdot pr_{\mathfrak{S}}(c_0 \to_{t_1} c_1) \cdot \ldots \cdot pr_{\mathfrak{S}}(c_{n-1} \to_{t_n} c_n)$. Here, all paths f which are not "admissible" (e.g., guards are not fulfilled, transitions are starting or ending in wrong locations, etc.) have probability $pr_{\mathfrak{S},\sigma_0}(f) = 0$.

The semantics of PIPs can be defined via a corresponding probability space, obtained by a standard cylinder construction. Let $\mathbb{P}_{\mathfrak{S},\sigma_0}$ denote the probability measure which lifts $pr_{\mathfrak{S},\sigma_0}$ to cylinder sets: For any $f \in \mathsf{Path}$, we have $pr_{\mathfrak{S},\sigma_0}(f) = \mathbb{P}_{\mathfrak{S},\sigma_0}(\mathrm{Pre}_f)$ for the set Pre_f of all infinite runs with prefix f. So $\mathbb{P}_{\mathfrak{S},\sigma_0}(\Theta)$ is the probability that a run from $\Theta \subseteq \mathsf{Run}$ is obtained when using the scheduler $\mathfrak{S}$ and starting in σ_0. Let $\mathbb{E}_{\mathfrak{S},\sigma_0}$ denote the associated expected value operator.

So for any random variable $X : \mathsf{Run} \to \overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, we have $\mathbb{E}_{\mathfrak{S},\sigma_0}(X) = \sum_{n \in \overline{\mathbb{N}}} n \cdot \mathbb{P}_{\mathfrak{S},\sigma_0}(X = n)$. For a detailed construction, see [26].

Definition 2 (Expected Runtime). *For $g \in \mathcal{GT}$, $\mathcal{R}_g : \mathsf{Run} \to \overline{\mathbb{N}}$ is a random variable with $\mathcal{R}_g(c_0 \to_{t_1} c_1 \to_{t_2} \cdots) = |\{i \in \mathbb{N} \mid t_i \in g\}|$, i.e., $\mathcal{R}_g(\vartheta)$ is the number of times that a transition from g was applied in the run $\vartheta \in \mathsf{Run}$. Moreover, the random variable $\mathcal{R} : \mathsf{Run} \to \overline{\mathbb{N}}$ denotes the number of transitions that were executed before termination, i.e., for all $\vartheta \in \mathsf{Run}$ we have $\mathcal{R}(\vartheta) = \sum_{g \in \mathcal{GT}} \mathcal{R}_g(\vartheta)$. For a scheduler $\mathfrak{S}$ and $\sigma_0 \in \Sigma$, the* expected runtime *of g is $\mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R}_g)$ and the* expected runtime *of the program is $\mathcal{R}_{\mathfrak{S},\sigma_0} = \mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R})$.*

The goal of complexity analysis for a PIP is to compute a bound on its *expected runtime complexity*. The set of *bounds* $\mathcal{B}$ consists of all functions from $\Sigma \to \mathbb{R}_{\geq 0}$.

Definition 3 (Expected Runtime Bound and Complexity [27]). *The function $\mathcal{RB} : \mathcal{GT} \to \mathcal{B}$ is an* expected runtime bound *if $(\mathcal{RB}(g))(\sigma_0) \geq \sup_{\mathfrak{S}} \mathbb{E}_{\mathfrak{S},\sigma_0}(\mathcal{R}_g)$ for all $\sigma_0 \in \Sigma$ and all $g \in \mathcal{GT}$. Then $\sum_{g \in \mathcal{GT}} \mathcal{RB}(g)$ is a bound on the* expected runtime complexity *of the whole program, i.e., $\sum_{g \in \mathcal{GT}}((\mathcal{RB}(g))(\sigma_0)) \geq \sup_{\mathfrak{S}} \mathcal{R}_{\mathfrak{S},\sigma_0}$ for all $\sigma_0 \in \Sigma$.*

3 Control-Flow Refinement for PIPs

We now introduce our novel CFR algorithm for *probabilistic* integer programs, based on the partial evaluation technique for non-probabilistic programs from [13, 14, 18]. In particular, our algorithm coincides with the classical CFR technique when the program is non-probabilistic. The goal of CFR is to transform a program $\mathcal{P}$ into a program $\mathcal{P}'$ which is "easier" to analyze. Thm. 4 shows the soundness of our approach, i.e., that $\mathcal{P}$ and $\mathcal{P}'$ have the same expected runtime complexity.

Our CFR technique considers "abstract" evaluations which operate on sets of states. These sets are characterized by conjunctions τ of constraints from $\mathcal{C}(\mathcal{PV})$, i.e., τ stands for all states $\sigma \in \Sigma$ with $\sigma \models \tau$. We now label locations ℓ by formulas τ which describe (a superset of) those states σ which can occur in ℓ, i.e., where a configuration (ℓ, σ) is reachable from some initial configuration (ℓ_0, σ_0). We begin with labeling every location by the constraint `true`. Then we add new copies of the locations with refined labels τ by considering how the updates of transitions affect the constraints of their start locations and their guards. The labeled locations become the new locations in the refined program.

Since a location might be reachable by different paths, we may construct several variants $\langle \ell, \tau_1 \rangle, \ldots, \langle \ell, \tau_n \rangle$ of the same original location ℓ. Thus, the formulas τ are not necessarily invariants that hold for *all* evaluations that reach a location ℓ, but we perform a case analysis and split up a location ℓ according to the different sets of states that may reach ℓ. Our approach ensures that a labeled location $\langle \ell, \tau \rangle$ can only be reached by configurations (ℓ, σ) where $\sigma \models \tau$.

We apply CFR only *on-demand* on a (sub)set of transitions $\mathcal{S} \subseteq \mathcal{T}$ (thus, CFR can be performed in a *modular* way for different subsets $\mathcal{S}$). In practice, we choose $\mathcal{S}$ heuristically and use CFR only on transitions where our currently inferred

runtime bounds are "not yet good enough". Then, for $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{GT})$, the result of the CFR algorithm is the program $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \langle \ell_0, \texttt{true} \rangle, \mathcal{GT}')$ where $\mathcal{L}'$ and $\mathcal{GT}'$ are the smallest sets satisfying the properties (3), (4), and (5) below.

First, we require that for all $\ell \in \mathcal{L}$, all "original" locations $\langle \ell, \texttt{true} \rangle$ are in $\mathcal{L}'$. In these locations, we do not have any information on the possible states yet:

$$\forall\, \ell \in \mathcal{L}.\ \langle \ell, \texttt{true} \rangle \in \mathcal{L}' \tag{3}$$

If we already introduced a location $\langle \ell, \tau \rangle \in \mathcal{L}'$ and there is a transition $(\ell, \varphi, p, \eta, \ell') \in \mathcal{S}$, then (4) requires that we also add the location $\langle \ell', \tau_{\varphi,\eta,\ell'} \rangle$ to $\mathcal{L}'$. The formula $\tau_{\varphi,\eta,\ell'}$ over-approximates the set of states that can result from states that satisfy τ and the guard φ of the transition when applying the update η. More precisely, $\tau_{\varphi,\eta,\ell'}$ has to satisfy $(\tau \wedge \varphi) \models \eta(\tau_{\varphi,\eta,\ell'})$. For example, if $\tau = (x = 0)$, $\varphi = \texttt{true}$, and $\eta(x) = x - 1$, then we might have $\tau_{\varphi,\eta,\ell'} = (x = -1)$.

To ensure that every $\ell' \in \mathcal{L}$ only gives rise to *finitely* many new labeled locations $\langle \ell', \tau_{\varphi,\eta,\ell'} \rangle$, we perform *property-based abstraction*: For every location ℓ', we use a finite so-called *abstraction layer* $\alpha_{\ell'} \subset \{p_1 \sim p_2 \mid p_1, p_2 \in \mathbb{Z}[\mathcal{PV}]$ and $\sim \in \{<, \leq, =\}\}$ (see [14] for heuristics to compute $\alpha_{\ell'}$). Then we require that $\tau_{\varphi,\eta,\ell'}$ must be a conjunction of constraints from $\alpha_{\ell'}$ (i.e., $\tau_{\varphi,\eta,\ell'} \subseteq \alpha_{\ell'}$ when regarding sets of constraints as their conjunction). This guarantees termination of our CFR algorithm, since for every location ℓ' there are only finitely many possible labels.

$$\begin{aligned} \forall\, \langle \ell, \tau \rangle \in \mathcal{L}'.\ \forall\, (\ell, \varphi, p, \eta, \ell') \in \mathcal{S}.\ \langle \ell', \tau_{\varphi,\eta,\ell'} \rangle \in \mathcal{L}' \\ \text{where } \tau_{\varphi,\eta,\ell'} = \{\psi \in \alpha_{\ell'} \mid (\tau \wedge \varphi) \models \eta(\psi)\} \end{aligned} \tag{4}$$

Finally, we have to ensure that $\mathcal{GT}'$ contains all "necessary" (general) transitions. To this end, we consider all $g \in \mathcal{GT}$. The transitions $(\ell, \varphi, p, \eta, \ell')$ in $g \cap \mathcal{S}$ now have to connect the appropriately labeled locations. Thus, for all labeled variants $\langle \ell, \tau \rangle \in \mathcal{L}'$, we add the transition $(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \tau_{\varphi,\eta,\ell'} \rangle)$. In contrast, the transitions $(\ell, \varphi, p, \eta, \ell')$ in $g \setminus \mathcal{S}$ only reach the location where ℓ' is labeled with $\texttt{true}$, i.e., here we add the transition $(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \texttt{true} \rangle)$.

$$\begin{aligned} &\forall\, \langle \ell, \tau \rangle \in \mathcal{L}'.\ \forall\, g \in \mathcal{GT}. \\ &(\{(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \tau_{\varphi,\eta,\ell'} \rangle) \mid (\ell, \varphi, p, \eta, \ell') \in g \cap \mathcal{S}\} \cup \\ &\ \{(\langle \ell, \tau \rangle, \tau \wedge \varphi, p, \eta, \langle \ell', \texttt{true} \rangle) \mid (\ell, \varphi, p, \eta, \ell') \in g \setminus \mathcal{S}\}) \qquad \in \mathcal{GT}' \end{aligned} \tag{5}$$

$\mathcal{L}'$ and $\bigcup_{g \in \mathcal{GT}'} g$ are finite due to the property-based abstraction, as there are only finitely many possible labels for each location. Hence, repeatedly "unrolling" transitions by (5) leads to the (unique) least fixpoint. Moreover, (5) yields proper general transitions, i.e., their probabilities still add up to 1. In practice, we remove transitions with unsatisfiable guards, and locations that are not reachable from $\langle \ell_0, \texttt{true} \rangle$. Thm. 4 shows the soundness of our approach (see [26] for its proof).

Theorem 4 (Soundness of CFR for PIPs). *Let $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \langle \ell_0, \texttt{true} \rangle, \mathcal{GT}')$ be the PIP such that $\mathcal{L}'$ and $\mathcal{GT}'$ are the smallest sets satisfying (3), (4), and (5). Let $\mathcal{R}^{\mathcal{P}}_{\mathfrak{S},\sigma_0}$ and $\mathcal{R}^{\mathcal{P}'}_{\mathfrak{S},\sigma_0}$ be the expected runtimes of $\mathcal{P}$ and $\mathcal{P}'$, respectively. Then for all $\sigma_0 \in \Sigma$ we have $\sup_{\mathfrak{S}} \mathcal{R}^{\mathcal{P}}_{\mathfrak{S},\sigma_0} = \sup_{\mathfrak{S}} \mathcal{R}^{\mathcal{P}'}_{\mathfrak{S},\sigma_0}$.*

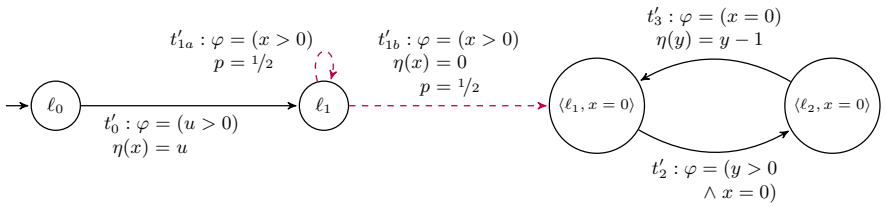

Fig. 2: Result of Control-Flow Refinement with $\mathcal{S} = \{t_{1a}, t_{1b}, t_2, t_3\}$

CFR Algorithm and its Runtime: To implement the fixpoint construction of Thm. 4 (i.e., to compute the PIP $\mathcal{P}'$), our algorithm starts by introducing all "original" locations $\langle \ell, \texttt{true} \rangle$ for $\ell \in \mathcal{L}$ according to (3). Then it iterates over all labeled locations $\langle \ell, \tau \rangle$ and all transitions $t \in \mathcal{T}$. If the start location of t is ℓ, then the algorithm extends $\mathcal{GT}'$ by a new transition according to (5). Moreover, it also adds the corresponding labeled target location to $\mathcal{L}'$ (as in (4)), if $\mathcal{L}'$ did not contain this labeled location yet. Afterwards, we mark $\langle \ell, \tau \rangle$ as finished and proceed with a previously computed labeled location that is not marked yet. So our implementation iteratively "unrolls" transitions by (5) until no new labeled locations are obtained (this yields the least fixpoint mentioned above). Thus, unrolling steps with transitions from $\mathcal{T} \setminus \mathcal{S}$ do not invoke further computations.

To over-approximate the runtime of this algorithm, note that for every location $\ell \in \mathcal{L}$, there can be at most $2^{|\alpha_\ell|}$ many labeled locations of the form $\langle \ell, \tau \rangle$. So if $\mathcal{L} = \{\ell_0, \ldots, \ell_n\}$, then the overall number of labeled locations is at most $2^{|\alpha_{\ell_0}|} + \ldots + 2^{|\alpha_{\ell_n}|}$. Hence, the algorithm performs at most $|\mathcal{T}| \cdot (2^{|\alpha_{\ell_0}|} + \ldots + 2^{|\alpha_{\ell_n}|})$ unrolling steps.

Example 5. For the PIP in Fig. 1 and $\mathcal{S} = \{t_{1a}, t_{1b}, t_2, t_3\}$, by (3) we start with $\mathcal{L}' = \{\langle \ell_i, \texttt{true} \rangle \mid i \in \{0, 1, 2\}\}$. We abbreviate $\langle \ell_i, \texttt{true} \rangle$ by ℓ_i in the final result of the CFR algorithm in Fig. 2. As $t_0 \in \{t_0\} \setminus \mathcal{S}$, by (5) t_0 is redirected such that it starts at $\langle \ell_0, \texttt{true} \rangle$ and ends in $\langle \ell_1, \texttt{true} \rangle$, resulting in t'_0. We always use primes to indicate the correspondence between new and original transitions.

Next, we consider $\{t_{1a}, t_{1b}\} \subseteq \mathcal{S}$ with the guard $\varphi = (x > 0)$ and start location $\langle \ell_1, \texttt{true} \rangle$. We first handle t_{1a} which has the update $\eta = \mathsf{id}$. We use the abstraction layer $\alpha_{\ell_0} = \varnothing$, $\alpha_{\ell_1} = \{x = 0\}$, and $\alpha_{\ell_2} = \{x = 0\}$. Thus, we have to find all $\psi \in \alpha_{\ell_1} = \{x = 0\}$ such that $(\texttt{true} \wedge x > 0) \models \eta(\psi)$. Hence, $\tau_{x>0,\mathsf{id},\ell_1}$ is the empty conjunction $\texttt{true}$ as no ψ from α_{ℓ_1} satisfies this property. We obtain

$$t'_{1a} : (\langle \ell_1, \texttt{true} \rangle, x > 0, 1/2, \mathsf{id}, \langle \ell_1, \texttt{true} \rangle).$$

In contrast, t_{1b} has the update $\eta(x) = 0$. To determine $\tau_{x>0,\eta,\ell_1}$, again we have to find all $\psi \in \alpha_{\ell_1} = \{x = 0\}$ such that $(\texttt{true} \wedge x > 0) \models \eta(\psi)$. Here, we get $\tau_{x>0,\eta,\ell_1} = (x = 0)$. Thus, by (4) we create the location $\langle \ell_1, x = 0 \rangle$ and obtain

$$t'_{1b} : (\langle \ell_1, \texttt{true} \rangle, x > 0, 1/2, \eta(x) = 0, \langle \ell_1, x = 0 \rangle).$$

As t_{1a} and t_{1b} form one general transition, by (5) we obtain $\{t'_{1a}, t'_{1b}\} \in \mathcal{GT}'$.

Now, we consider transitions resulting from $\{t_{1a}, t_{1b}\}$ with the start location $\langle \ell_1, x = 0 \rangle$. However, $\tau = (x = 0)$ and the guard $\varphi = (x > 0)$ are conflicting, i.e., the transitions would have an unsatisfiable guard $\tau \wedge \varphi$ and are thus omitted.

Next, we consider transitions resulting from t_2 with $\langle \ell_1, \texttt{true} \rangle$ or $\langle \ell_1, x = 0 \rangle$ as their start location. Here, we obtain two (general) transitions $\{t_2'\}, \{t_2''\} \in \mathcal{GT}'$:

$$t_2' : (\langle \ell_1, x = 0 \rangle, y > 0 \wedge x = 0, 1, \mathsf{id}, \langle \ell_2, x = 0 \rangle)$$
$$t_2'' : (\langle \ell_1, \texttt{true} \rangle, y > 0 \wedge x = 0, 1, \mathsf{id}, \langle \ell_2, x = 0 \rangle)$$

However, t_2'' can be ignored since $x = 0$ contradicts the invariant $x > 0$ at $\langle \ell_1, \texttt{true} \rangle$. KoAT uses Apron [20] to infer invariants like $x > 0$ automatically. Finally, t_3 leads to the transition $t_3' : (\langle \ell_2, x = 0 \rangle, x = 0, 1, \eta(y) = y - 1, \langle \ell_1, x = 0 \rangle)$. Thus, we obtain $\mathcal{L}' = \{\langle \ell_i, \texttt{true} \rangle \mid i \in \{0, 1\}\} \cup \{\langle \ell_i, x = 0 \rangle \mid i \in \{1, 2\}\}$.

KoAT infers a bound $\mathcal{RB}(g)$ for each $g \in \mathcal{GT}$ individually (thus, non-probabilistic program parts can be analyzed by classical techniques). Then $\sum_{g \in \mathcal{GT}} \mathcal{RB}(g)$ is a bound on the expected runtime complexity of the whole program, see Def. 3.

Example 6. We now infer a bound on the expected runtime complexity of the PIP in Fig. 2. Transition t_0' is not on a cycle, i.e., it can be evaluated at most once. So $\mathcal{RB}(\{t_0'\}) = 1$ is an (expected) runtime bound for the general transition $\{t_0'\}$.

For the general transition $\{t_{1a}', t_{1a}'\}$, KoAT infers the expected runtime bound 2 via probabilistic linear ranking functions (PLRFs, see e.g., [27]). More precisely, KoAT finds the *constant* PLRF $\{\ell_1 \mapsto 2, \langle \ell_1, x = 0 \rangle \mapsto 0\}$. In contrast, in the original program of Fig. 1, $\{t_{1a}, t_{1b}\}$ is not decreasing w.r.t. any constant PLRF, because t_{1a} and t_{1b} have the same target location. So here, every PLRF where $\{t_{1a}, t_{1b}\}$ decreases in expectation depends on x. However, such PLRFs do not yield a finite runtime bound in the end, as t_0 instantiates x by the non-deterministic value u. Therefore, KoAT fails on the program of Fig. 1 without using CFR.

For the program of Fig. 2, KoAT infers $\mathcal{RB}(\{t_2'\}) = \mathcal{RB}(\{t_3'\}) = y$. By adding all runtime bounds, we obtain the bound $3 + 2 \cdot y$ on the expected runtime complexity of the program in Fig. 2 and thus by Thm. 4 also of the program in Fig. 1.

4 Implementation, Evaluation, and Conclusion

We presented a novel control-flow refinement technique for probabilistic programs and proved that it does not modify the program's expected runtime complexity. This allows us to combine CFR with approaches for complexity analysis of probabilistic programs. Compared to its variant for non-probabilistic programs, the soundness proof of Thm. 4 for probabilistic programs is considerably more involved.

Up to now, our complexity analyzer KoAT used the tool iRankFinder [13] for CFR of non-probabilistic programs [18]. To demonstrate the benefits of CFR for complexity analysis of probabilistic programs, we now replaced the call to iRankFinder in KoAT by a native implementation of our new CFR algorithm. KoAT

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \omega$	AVG$^+$(s)	AVG(s)
KoAT+CFR	11 (2)	56 (12)	14	2	1	84 (14)	11.68	11.34
KoAT	9	41 (1)	16 (1)	2	1	69 (2)	2.71	2.41
Absynth	7	35	9	0	0	51	2.86	37.48
eco-imp	8	35	6	0	0	49	0.34	68.02

Table 1: Evaluation of CFR on Probabilistic Programs

is written in OCaml and it uses Z3 [12] for SMT solving, Apron [20] to generate invariants, and the Parma Polyhedra Library [8] for computations with polyhedra.

We used all 75 probabilistic benchmarks from [27, 29] and added 15 new benchmarks including our leading example and problems adapted from the *Termination Problem Data Base* [33], e.g., a probabilistic version of McCarthy's 91 function. Our benchmarks also contain examples where CFR is useful even if it cannot separate probabilistic from non-probabilistic program parts as in our leading example.

Table 1 shows the results of our experiments. We compared the configuration of KoAT with CFR ("KoAT+CFR") against KoAT without CFR. Moreover, as in [27], we also compared with the main other recent tools for inferring upper bounds on the expected runtimes of probabilistic integer programs (Absynth [29] and eco-imp [7]). As in the *Termination Competition* [17], we used a timeout of 5 minutes per example. The first entry in every cell is the number of benchmarks for which the tool inferred the respective bound. In brackets, we give the corresponding number when only regarding our new examples. For example, KoAT+CFR finds a finite expected runtime bound for 84 of the 90 examples. A linear expected bound (i.e., in $\mathcal{O}(n)$) is found for 56 of these 84 examples, where 12 of these benchmarks are from our new set. AVG(s) is the average runtime in seconds on all benchmarks and AVG$^+$(s) is the average runtime on all successful runs.

The experiments show that similar to its benefits for non-probabilistic programs [18], CFR also increases the power of automated complexity analysis for probabilistic programs substantially, while the runtime of the analyzer may become longer since CFR increases the size of the program. The experiments also indicate that a related CFR technique is not available in the other complexity analyzers. Thus, we conjecture that other tools for complexity or termination analysis of PIPs would also benefit from the integration of our CFR technique.

KoAT's source code, a binary, and a Docker image are available at:

https://koat.verify.rwth-aachen.de/prob_cfr

The website also explains how to use our CFR implementation separately (without the rest of KoAT), in order to access it as a black box by other tools. Moreover, the website provides a *web interface* to directly run KoAT online, and details on our experiments, including our benchmark collection.

Acknowledgements: We thank Yoann Kehler for helping with the implementation of our CFR technique in KoAT.

References

[1] S. Agrawal, K. Chatterjee, and P. Novotný. "Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs". In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158122.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Proc. SAS*. LNCS 5079. 2008, pp. 221–237. DOI: 10.1007/978-3-540-69166-2_15.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Object-Oriented Bytecode Programs". In: *Theor. Comput. Sci.* 413.1 (2012), pp. 142–159. DOI: 10.1016/j.tcs.2011.07.009.

[4] E. Albert, M. Bofill, C. Borralleras, E. Martín-Martín, and A. Rubio. "Resource Analysis driven by (Conditional) Termination Proofs". In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 722–739. DOI: 10.1017/S1471068419000152.

[5] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. "Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs". In: *Proc. SAS*. LNCS 6337. 2010, pp. 117–133. DOI: 10.1007/978-3-642-15769-1_8.

[6] M. Avanzini and G. Moser. "A Combination Framework for Complexity". In: *Proc. RTA*. LIPIcs 21. 2013, pp. 55–70. DOI: 10.4230/LIPIcs.RTA.2013.55.

[7] M. Avanzini, G. Moser, and M. Schaper. "A Modular Cost Analysis for Probabilistic Programs". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). URL: https://doi.org/10.1145/3428240.

[8] R. Bagnara, P. M. Hill, and E. Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Sci. Comput. Program.* 72 (2008), pp. 3–21. DOI: 10.1016/j.scico.2007.08.001.

[9] K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, and L. Verscht. "A Calculus for Amortized Expected Runtimes". In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571260.

[10] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. "Analyzing Runtime and Size Complexity of Integer Programs". In: *ACM Trans. Program. Lang. Syst.* 38 (2016), pp. 1–50. DOI: 10.1145/2866575.

[11] Q. Carbonneaux, J. Hoffmann, and Z. Shao. "Compositional Certified Resource Bounds". In: *Proc. PLDI*. 2015, pp. 467–478. DOI: 10.1145/2737924.2737955.

[12] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. TACAS*. LNCS 4963. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[13] J. J. Doménech and S. Genaim. "iRankFinder". In: *Proc. WST*. http://wst2018.webs.upv.es/wst2018proceedings.pdf. 2018, p. 83.

[14] J. J. Doménech, J. P. Gallagher, and S. Genaim. "Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis". In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

[15] A. Flores-Montoya. “Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations”. In: *Proc. FM*. LNCS 9995. 2016, pp. 254–273. DOI: 10.1007/978-3-319-48989-6_16.

[16] F. Frohn and J. Giesl. “Complexity Analysis for Java with AProVE”. In: *Proc. iFM*. LNCS 10510. 2017, pp. 85–101. DOI: 10.1007/978-3-319-66845-1_6.

[17] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. “The Termination and Complexity Competition”. In: *Proc. TACAS*. LNCS 11429. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[18] J. Giesl, N. Lommen, M. Hark, and F. Meyer. “Improving Automatic Complexity Analysis of Integer Programs”. In: *The Logic of Software. A Tasting Menu of Formal Methods*. LNCS 13360. 2022, pp. 193–228. DOI: 10.1007/978-3-031-08166-8_10.

[19] J. Hoffmann, A. Das, and S.-C. Weng. “Towards Automatic Resource Bound Analysis for OCaml”. In: *Proc. POPL*. 2017, pp. 359–373. DOI: 10.1145/3009837.3009842.

[20] B. Jeannet and A. Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *Proc. CAV*. LNCS 5643. 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[21] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. “Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms”. In: *J. ACM* 65 (2018), pp. 1–68. DOI: 10.1145/3208102.

[22] B. L. Kaminski, J. Katoen, and C. Matheja. “Expected Runtime Analyis by Program Verification”. In: *Foundations of Probabilistic Programming*. Ed. by G. Barthe, J. Katoen, and A. Silva. Cambridge University Press, 2020, 185–220. DOI: 10.1017/9781108770750.007.

[23] L. Leutgeb, G. Moser, and F. Zuleger. “Automated Expected Amortised Cost Analysis of Probabilistic Data Structures”. In: *Proc. CAV*. LNCS 13372. 2022, pp. 70–91. DOI: 10.1007/978-3-031-13188-2_4.

[24] N. Lommen, F. Meyer, and J. Giesl. “Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops”. In: *Proc. IJCAR*. LNCS 13385. 2022, pp. 734–754. DOI: 10.1007/978-3-031-10769-6_43.

[25] N. Lommen and J. Giesl. “Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs”. In: *Proc. FroCoS*. LNCS 14279. 2023, pp. 3–22. DOI: 10.1007/978-3-031-43369-6_1.

[26] N. Lommen, E. Meyer, and J. Giesl. “Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT”. In: *CoRR* abs/2402.03891 (2024). DOI: 10.48550/arXiv.2402.03891.

[27] F. Meyer, M. Hark, and J. Giesl. “Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes”. In: *Proc. TACAS*. LNCS 12651. 2021, pp. 250–269. DOI: 10.1007/978-3-030-72016-2_14.

[28] G. Moser and M. Schaper. “From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation”. In: *Inf. Comput.* 261 (2018), pp. 116–143. DOI: 10.1016/j.ic.2018.05.007.

[29] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. "Bounded Expectations: Resource Analysis for Probabilistic Programs". In: *Proc. PLDI*. 2018, pp. 496–512. URL: https://doi.org/10.1145/3192366.3192394.

[30] L. Noschinski, F. Emmes, and J. Giesl. "Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs". In: *J. Autom. Reason.* 51 (2013), pp. 27–56. DOI: 10.1007/s10817-013-9277-6.

[31] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, and C. Matheja. "A Deductive Verification Infrastructure for Probabilistic Programs". In: *Proc. ACM Program. Lang.* 7.OOPSLA (2023), pp. 2052–2082. DOI: 10.1145/3622870.

[32] M. Sinn, F. Zuleger, and H. Veith. "Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints". In: *J. Autom. Reason.* 59.1 (2017), pp. 3–45. DOI: 10.1007/s10817-016-9402-4.

[33] TPDB (Termination Problem Data Base). URL: https://github.com/TermCOMP/TPDB.

[34] D. Wang, D. M. Kahn, and J. Hoffmann. "Raising Expectations: Automating Expected Cost Analysis with Types". In: *Proc. ACM Program. Lang.* 4.ICFP (2020). URL: https://doi.org/10.1145/3408992.