KoAT: Automatic Complexity and Termination Analysis of Integer Programs⋆

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

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

Abstract. KoAT is a tool to automatically infer complexity bounds and prove termination of (possibly recursive) integer programs. To this end, KoAT implements an alternating modular inference of upper runtime and size bounds for program parts. In particular, KoAT uses a portfolio of different techniques to analyze subprograms. The power of our approach is demonstrated by an extensive experimental evaluation.

1 Introduction

Ensuring safety, security, and efficiency of software systems is a central goal of formal methods. While functional verification is used to prove that programs compute correct results, it is also crucial to analyze their resource consumption, such as worst-case execution time and termination. Since complex software is often abstracted into simplified integer programs during verification, inferring runtime and resource bounds for such integer programs is an important task in automated program analysis.

There exist numerous approaches to analyze complexity of programs on integers automatically, e.g., [1–4, 6, 14, 15, 20, 21, 23, 26, 31–33, 35, 47, 53, 55, 56]. We developed a *modular* technique for complexity analysis of programs with built-in integers and implemented it in the complexity analysis tool KoAT. It automatically infers runtime bounds for integer programs possibly consisting of multiple loops by handling some subprograms as *twn*-loops ("triangular weakly non-linear loops", where there exist "complete" techniques for analyzing termination and complexity) [27, 28, 40, 41, 45] and by using linear or multiphase-linear ranking functions (MΦRFs) for other subprograms [5–7, 14, 25, 29, 44, 54]. To lift local runtime bounds for isolated subprograms to bounds within the full program, we also infer size bounds on the values of the variables before entering such a subprogram. By inferring bounds for one subprogram after the other, in the end we obtain a bound on the runtime of the whole program. Moreover, besides its application for complexity analysis, we recently extended KoAT by a specific mode in order to prove termination of integer programs. While KoAT can also analyze probabilistic programs [42, 49], here we focus on non-probabilistic programs to ease the presentation.

⋆ funded by the DFG Research Training Group 2236 UnRAVeL

while $(x_1 > 0)$ **do**
  $(x_1, x_2, x_3) \leftarrow (x_1 - 1, x_1, 1)$
  while $(x_2 > x_3 \wedge x_2 > 0)$ **do** $(x_2, x_3) \leftarrow (2 \cdot x_2,\ 3 \cdot x_3)$ **end**
end

Fig. 1: Program with two Nested Loops

Example 1. Consider the program in Fig. 1 which consists of two nested **while**-loops. The outer **while**-loop is executed at most x_1 times. If one just considers the inner **while**-loop in isolation, then it is executed at most $\log_2(x_2) + 2$ times (see Sect. 2.2). This runtime bound can be obtained by our technique based on *twn*-loops, but not by linear ranking functions. Note that x_2's value is bounded by x_1 before executing the inner loop, where "x_1" refers to the initial value of the program variable. This observation is crucial to infer the (non-isolated) runtime of the inner loop since it is executed at most $x_1 \cdot (\log_2(\text{size}(x_2)) + 2) = x_1 \cdot (\log_2(x_1) + 2)$ times, where "size(x_2)" denotes an over-approximation of the (absolute) value of x_2 before entering the inner loop.

Structure. In Sect. 2.1, we formalize our notion of integer programs. Afterwards, we sketch the techniques implemented in KoAT to analyze such integer programs in Sect. 2.2. In Sect. 3 we discuss related work. We conclude with an experimental evaluation in Sect. 4.

2 Complexity and Termination Analysis of Integer Programs

In this section, we present KoAT's modular approach for the automatic analysis of runtime complexity and termination of integer programs. More precisely, in Sect. 2.1, we introduce our formalism of integer programs. In Sect. 2.2, we then sketch the techniques used to analyze these programs.

2.1 Integer Programs

For integer programs, we use a formalism based on transitions. As usual, $\mathbb{Z}[\mathcal{V}]$ denotes the polynomial ring with variables $\mathcal{V}$ and integer coefficients. $\mathcal{F}(\mathcal{V})$ denotes the set of formulas, where a formula is a propositional combination of polynomial (in)equations over the variables $\mathcal{V}$ (i.e., expressions like $x + y \le 2 \wedge x^2 = 2$).

Definition 2 (Integer Program). $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ *is an* integer program *where*

- $\mathcal{V}$ *is a finite set of (program)* variables*,*
- $\mathcal{L}$ *is a finite set of* locations *with an* initial location $\ell_0 \in \mathcal{L}$*, and*

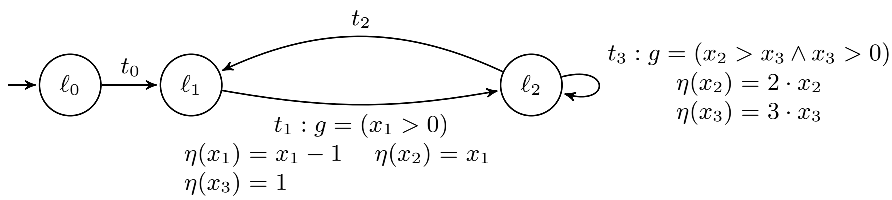

Fig. 2: An Integer Program Corresponding to Fig. 1

- $\mathcal{T}$ *is a finite set of* transitions. *A transition is a 4-tuple* (ℓ, g, η, ℓ') *with a* start location $\ell \in \mathcal{L}$*,* target location $\ell' \in \mathcal{L} \setminus \{\ell_0\}$*,* guard $g \in \mathcal{F}(\mathcal{V})$*, and* update function $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ *mapping variables to update polynomials.*

An integer program may have two kinds of non-determinism. Non-deterministic branching is realized by multiple transitions with the same start location whose guards are non-exclusive. Non-deterministic sampling can be modeled by temporary variables (which can be restricted in the guard of a transition). For simplicity, we omitted these temporary variables in Def. 2 (see [25, 40, 45] for the extended definition). Moreover, in [44] we extended our approach to integer programs with *function calls*, including non-tail recursion.

Example 3. In the integer program of Fig. 2 which corresponds to the pseudocode of Fig. 1, we omitted identity updates $\eta(v) = v$ and guards where g is `true`. Here, the variables are $\mathcal{V} = \{x_1, x_2, x_3\}$, and $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$ are the locations, where ℓ_0 is the initial location.

A *state* $\sigma : \mathcal{V} \to \mathbb{Z}$ is a mapping from variables to numbers. In the following, Σ denotes the set of all states. A *configuration* is an element of $\mathcal{L} \times \Sigma$. To define the semantics of integer programs, an evaluation step moves from one configuration $(\ell, \sigma) \in \mathcal{L} \times \Sigma$ to another configuration (ℓ', σ') via a transition (ℓ, g, η, ℓ') where the state $\sigma \in \Sigma$ satisfies the guard g (denoted by $\sigma(g) =$ `true`). Let $\sigma(\eta(v))$ denote the expression that results from replacing all variables x by $\sigma(x)$ in the polynomial $\eta(v)$. Then σ' is obtained by applying σ on the update η (i.e., $\sigma'(v) = \sigma(\eta(v))$ for all $v \in \mathcal{V}$). From now on, we fix a program $\mathcal{P} = (\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$.

Definition 4 (Evaluation of Integer Programs). *A* configuration *is an element of* $\mathcal{L} \times \Sigma$*. For two configurations* (ℓ, σ) *and* (ℓ', σ')*, and a transition* $t = (\ell_t, g, \eta, \ell'_t) \in \mathcal{T}$*,* $(\ell, \sigma) \to_t (\ell', \sigma')$ *is an* evaluation *step by* t *if*

- $\ell = \ell_t$ *and* $\ell' = \ell'_t$*,*
- $\sigma(g) =$ `true`*, and*
- *for every variable* $v \in \mathcal{V}$ *we have* $\sigma'(v) = \sigma(\eta(v))$*.*

We denote the union of all relations $\to_t$ *for* $t \in \mathcal{T}$ *by* $\to_{\mathcal{T}}$*. Moreover, we abbreviate* $(\ell_0, \sigma_0) \to (\ell_1, \sigma_1) \to \cdots \to (\ell_k, \sigma_k)$ *by* $(\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)$ *and write* $(\ell, \sigma) \to^*_{\mathcal{T}} (\ell', \sigma')$ *if* $(\ell, \sigma) \to^k_{\mathcal{T}} (\ell', \sigma')$ *for some* $k \in \mathbb{N}$*.*

Example 5. When denoting states $\sigma \in \Sigma$ as tuples $(\sigma(x_1), \sigma(x_2), \sigma(x_3)) \in \mathbb{Z}^3$, then a run of Fig. 2 that starts in $(5,0,0)$ has the form $(\ell_0,(5,0,0)) \to_{t_0} (\ell_1,(5,0,0)) \to_{t_1} (\ell_2,(4,5,1)) \to_{t_3}^4 (\ell_2,(4,80,81)) \to_{t_2} (\ell_1,(4,80,81)) \to \cdots$.

For an initial state $\sigma_0 \in \Sigma$, the *runtime complexity* $\mathrm{rc}(\sigma_0)$ of a program corresponds to the length of a longest evaluation starting in σ_0.

Definition 6 (Runtime Complexity). *The* runtime complexity *is* $\mathrm{rc}\colon \Sigma \to \overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$ *with* $\mathrm{rc}(\sigma_0) = \sup\left\{k \in \mathbb{N} \mid \exists(\ell',\sigma').\, (\ell_0,\sigma_0) \to_{\mathcal{T}}^k (\ell',\sigma')\right\}$. *An integer program is* terminating *if and only if* $\mathrm{rc}(\sigma_0) \neq \omega$ *for all* $\sigma_0 \in \Sigma$.

Example 7. For the program of Fig. 2 and the state $\sigma \in \Sigma$ from Ex. 5 with $\sigma(x_1) = 5$ and $\sigma(x_2) = \sigma(x_3) = 0$, we obtain the runtime complexity $\mathrm{rc}(\sigma) = 25$.

Now we define our notion of *bounds*. KoAT implements polynomial, logarithmic, and exponential bounds like x^2+y, $\log_2(x)$, or 2^x. We only consider bounds which correspond to functions f that are weakly monotonically increasing in all arguments, i.e., where $x \leq y$ implies $f(\dots x \dots) \leq f(\dots y \dots)$. In this way, if f and g are both bounds, then $f \circ g$ is also a bound, i.e., bounds can be "composed" easily. For example, we used this in the introductory Ex. 1 where we inserted the size bound "size(x_2)" into the runtime bound $\log_2(x_2) + 2$ of the second loop. In principle, every weakly monotonically increasing function could be used as a bound in our framework. However, in KoAT we restrict ourselves to bounds which are easy to represent and to compute with, and which cover the most prominent complexity classes.

Definition 8 (Bounds). *The set of* bounds $\mathcal{B}$ *is the smallest set containing all*

- *natural numbers* $\overline{\mathbb{N}} \subseteq \mathcal{B}$,
- *variables* $\mathcal{V} \subseteq \mathcal{B}$, *and*
- $\{b_1 + b_2, \max(b_1,b_2), b_1 \cdot b_2, p^{b_1}, \log_k(b_1)\} \subseteq \mathcal{B}$ *for all* $b_1, b_2 \in \mathcal{B}$, *all polynomials* $p \in \mathbb{N}[\mathcal{V}]$, *and all* $k \in \mathbb{R}_{>1}$.[1]

We measure the size of variables by their absolute values. For a state $\sigma \in \Sigma$, $|\sigma|$ is the state with $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$ and $[\![b]\!]_{|\sigma|}$ denotes the evaluation of a bound $b \in \mathcal{B}$ at the state $|\sigma|$. So if $\sigma(x) = 5$ and $\sigma(y) = -4$, then we have

$$[\![x^2 + y]\!]_{|\sigma|} = |\sigma|(x)^2 + |\sigma|(y) = 25 + 4 = 29.$$

A function $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ is a *runtime bound* if for each transition t and initial state $\sigma_0 \in \Sigma$, $\mathcal{RB}(t)$ evaluated in the state $|\sigma_0|$ over-approximates the number of evaluation steps with t in any run starting in the configuration (ℓ_0, σ_0). Let $\to_{\mathcal{T}}^* \circ \to_t$ denote the relation where arbitrary many evaluation steps are followed by a step with the transition t.

[1] More precisely, instead of $\log_k(b_1)$ we use the function $\lceil \log_k(\max\{1, b_1\}) \rceil$ to ensure that bounds are well defined, weakly monotonically increasing, and evaluate to natural numbers.

Definition 9 (Runtime Bound). *The function* $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ *is a* runtime bound *if for all* $t \in \mathcal{T}$ *and all states* $\sigma_0 \in \Sigma$ *we have*

$$[\![\mathcal{RB}(t)]\!]_{|\sigma_0|} \;\geq\; \sup\left\{k \in \mathbb{N} \mid \exists\,(\ell', \sigma').\, (\ell_0, \sigma_0)\, (\to_{\mathcal{T}}^{*} \circ \to_t)^k\, (\ell', \sigma')\right\}.$$

Example 10. For the program of Fig. 2, a runtime bound is $\mathcal{RB}(t_0) = 1$ (as t_0 is not on a cycle), $\mathcal{RB}(t_1) = \mathcal{RB}(t_2) = x_1$ (this can be inferred by MΦRFs), and $\mathcal{RB}(t_3) = x_1 \cdot (\log_2(x_1) + 2)$ (this can be inferred by *twn*-loops), see Sect. 2.2. Now a runtime bound for the full program is the sum of all these expressions. For example, we obtain $[\![1 + x_1 \cdot \log_2(x_1) + 4 \cdot x_1]\!]_{|\sigma|} \approx 32.61 > 25 = \mathrm{rc}(\sigma)$ for the state $\sigma = (5, 0, 0)$ from Ex. 5 and 7.

In the following, we define size bounds for variables v after evaluating a transition $t \in \mathcal{T}$: $\mathcal{SB}(t, v)$ is a *size bound* for the variable v w.r.t. the transition t if for any run starting in the initial state $\sigma_0 \in \Sigma$, $[\![\mathcal{SB}(t, v)]\!]_{|\sigma_0|}$ is greater or equal to the largest absolute value of v after evaluating t.

Definition 11 (Size Bounds). *A function* $\mathcal{SB} : (\mathcal{T} \times \mathcal{V}) \to \mathcal{B}$ *is a* size bound *if for all* $(t, v) \in \mathcal{T} \times \mathcal{V}$ *and all states* $\sigma_0 \in \Sigma$ *we have*

$$[\![\mathcal{SB}(t, v)]\!]_{|\sigma_0|} \geq \sup\left\{|\sigma'(v)| \mid \exists\, \ell' \in \mathcal{L}.\, (\ell_0, \sigma_0)\, (\to_{\mathcal{T}}^{*} \circ \to_t)\, (\ell', \sigma')\right\}.$$

Example 12. For example, a size bound for t_1 and x_2 in Fig. 2 is $\mathcal{SB}(t_1, x_2) = x_1$. Such size bounds are needed for our modular analysis, e.g., to handle nested loops. The transition t_3 (with the location ℓ_2) can be considered as a subprogram of the full integer program, i.e., it corresponds to the inner loop of the program in Fig. 2. An evaluation in the subprogram $\{t_3\}$ can only make t_3-steps and a local runtime bound for this subprogram is $\log_2(x_2) + 2$ (see Sect. 2.2).

To consider evaluations in the full program of Fig. 2, we lift *local* to *global* runtime bounds. To this end, we need size bounds. As in Def. 9, a global runtime bound for t_3 over-approximates how often t_3 can be applied in a run of the full program. To obtain a global runtime bound from a local runtime bound in the subprogram $\{t_3\}$, (i) we must consider how often the subprogram $\{t_3\}$ is entered (at most $\mathcal{RB}(t_1)$ times) and (ii) we must consider the sizes of the variables upon entry into the subprogram (i.e., at most $\mathcal{SB}(t_1, v)$ for all $v \in \mathcal{V}$). So, Problem (i) is captured by the runtime bounds of the *entry transitions* (in our case only t_1 is an entry transition of the subprogram $\{t_3\}$). Problem (ii) is handled via size bounds for the entry transitions (i.e., by replacing x_2 with $\mathcal{SB}(t_1, x_2)$ in the local runtime bound $\log_2(x_2) + 2$). Thus, for our example, we obtain

$$\mathcal{RB}(t_3) = \mathcal{RB}(t_1) \cdot (\log_2(\mathcal{SB}(t_1, x_2)) + 2) = x_1 \cdot (\log_2(x_1) + 2).$$

2.2 Analyzing Integer Programs

We now sketch our modular approach to analyze integer programs. An overview of our tool KoAT is depicted in Fig. 3. In order to compute runtime and size bounds, we analyze subprograms in topological order, i.e., in case of multiple

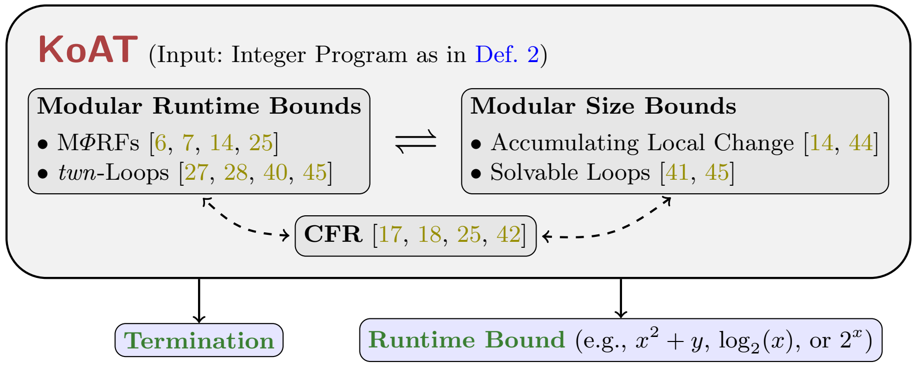

Fig. 3: KoAT: Automatic Complexity and Termination Analysis of Integer Programs

consecutive loops, we start with the first loop and propagate knowledge about the resulting values of variables to subsequent loops. By handling one subprogram after the other, in the end we obtain a bound on the runtime complexity of the whole program. The approach uses size bounds to infer runtime bounds, i.e., size bounds on the entry transitions of subprograms are used to lift local runtime bounds for isolated subprograms to global runtime bounds for the full program. Similarly, we also use runtime bounds to infer size bounds, since runtime bounds provide information on how often the local change of a transition can be performed. Hence, runtime and size bound computations are alternated until all bounds are finite or no bound can be improved further. Finally, if our analysis fails to infer a "good enough" runtime bound, then KoAT applies *control-flow refinement* (CFR) to the integer program [17, 18, 25, 42] in order to obtain a program that is easier to analyze. In the following, we describe KoAT's techniques to infer local runtime bounds (via MΦRFs or *twn*-loops), size bounds, and CFR in more detail.

Multiphase-Linear Ranking Functions (MΦRFs). KoAT uses individual ranking functions to infer local runtime bounds for different subprograms. A ranking function assigns an integer to each configuration $(\ell, \sigma) \in \mathcal{L} \times \Sigma$. The value of a ranking function must not increase in evaluation steps, and it has to decrease by at least one in every evaluation step where a specific transition is applied. Moreover, it must have a positive value before each such evaluation step.

Example 13. For the integer program in Fig. 2, KoAT uses MΦRFs to infer runtime bounds for the transitions of the outer loop. For example, we can infer a runtime bound for t_1 by considering the ranking function

$$f(\ell_0, \sigma) = f(\ell_1, \sigma) = f(\ell_2, \sigma) = \sigma(x_1).$$

The reason is that t_1 decreases this ranking function while the other transitions do not increase it. In contrast, MΦRFs cannot be used to obtain a runtime bound for the inner loop consisting just of transition t_3. Instead, this inner loop is analyzed by our technique for *twn*-loops. As discussed in Ex. 12, the local runtime bound for this inner *twn*-loop is then lifted to a global runtime bound for t_3 in the full program.

A (nested) MΦRF [6, 7] extends the idea of ranking functions and uses a mapping f_i for every "phase" $1 \leq i \leq d$ of a program. Before each evaluation step, the value of the last phase f_d is required to be positive. Moreover, for all i, the sum of f_{i-1} and f_i before the evaluation step must be larger than the function f_i after the update. We set $f_0(\ell, \sigma)$ to 0 for all configurations (ℓ, σ). Thus, f_1 must be decreasing with each update. If f_1 becomes negative, then

$$f_1(\ell, \sigma) + f_2(\ell, \sigma) < f_2(\ell, \sigma)$$

and thus, now f_2 has to be decreasing with every update, and so on until f_d becomes decreasing. The program eventually terminates, since f_d must take a positive number whenever the program can be executed further. Moreover, if all functions $f_1, \ldots, f_d$ are defined via linear polynomials, then this implies a linear runtime bound for the subprogram. We refer to [25] for further details and the formal definition of MΦRFs for subprograms.

KoAT only implements ranking functions via linear polynomials since these are easy to generate automatically [54], and since they are already quite powerful when combining them within MΦRFs. Moreover, in contrast to *twn*-loops, MΦRFs are applicable to arbitrary integer programs with linear arithmetic.

Triangular Weakly Non-Linear-Loops. As an alternative approach to infer local runtime bounds, we integrated a technique [27, 28] to analyze termination of *triangular weakly non-linear loops* (*twn*-loops) in KoAT [40, 41, 45]. This approach also allows us to analyze programs with *non-linear* arithmetic, while the automated generation of MΦRFs is essentially limited to programs with linear arithmetic. An example for a terminating *twn*-loop which corresponds to the subprogram $\{t_3\}$ from Fig. 2 is given by:

$$\textbf{while } (x_2 > x_3 \wedge x_3 > 0) \textbf{ do } (x_2, x_3) \leftarrow (2 \cdot x_2, 3 \cdot x_3) \quad \textbf{end} \tag{1}$$

This loop does not admit a MΦRF over $\mathbb{R}$ (see [29]). The guard of such *twn*-loops are propositional formulas over (possibly non-linear) polynomial inequations. The update is *triangular*, i.e., we can order the variables such that the update of any x_i does not depend on the variables $x_1, \ldots, x_{i-1}$ with smaller indices. So the restriction to triangular updates prohibits "cyclic dependencies" of variables (e.g., where the new values of x_1 and x_2 both depend on the old values of x_1 and x_2). From a practical point of view, the restriction to triangular loops seems quite natural. For example, in [22], 1511 polynomial loops were extracted from the *Termination Problems Data Base* (TPDB) [58], the benchmark collection which is used at the annual *Termination and Complexity Competition* (TermComp) [24], and only 26 of them were non-triangular. Furthermore, the update of a *twn*-loop is *weakly non-linear*, i.e., no variable x_i has a non-linear

```
while 0 < x do
    if y < z then
        y ← y + x
    else
        x ← x − 1
```

Fig. 4: Original Loop

```
while 0 < x ∧ y < z do
    y ← y + x
while 0 < x ∧ y ≥ z do
    x ← x − 1
```

Fig. 5: After Control-Flow Refinement

occurrence in its own update. With triangularity and weak non-linearity, one can compute a *closed form* which corresponds to applying the loop's update n times and which is suitable for (dis)proving termination and inferring runtime bounds. Using these closed forms, termination can be reduced to a formula over $\mathbb{Z}$ [28] (whose satisfiability is decidable for linear arithmetic and where SMT solvers often also prove (un)satisfiability in the non-linear case). Furthermore, the closed forms can be used to infer runtime bounds. More precisely, one can always compute a polynomial runtime bound for every terminating *twn*-loop. Moreover, these bounds are linear if the loop only contains linear arithmetic. So for linear *twn*-loops, termination is decidable and a linear runtime bound can always be computed (if the loop is terminating).

The bounds are even logarithmic if in addition, the coefficients of the variables in their own updates have pairwise different absolute values [45]. This is the case for the *twn*-loop (1), because the coefficients of x_2 and x_3 in their own updates $\eta(x_2)$ and $\eta(x_3)$ are 2 and 3, which are different (absolute) values. Thus, here we can infer the logarithmic runtime bound $\log_2(x_2) + 2$ [44, App. C]. An analogous criterion for logarithmic runtimes can also be obtained for non-linear *twn*-loops. For future work, it would be interesting to develop classes of loops where one can give tightness guarantees for the obtained runtime bounds.

Size Bounds. As outlined before, size bounds are an important ingredient of our modular approach for complexity analysis. Currently, we use two techniques to infer size bounds in KoAT. On the one hand, we integrated a procedure for *solvable* loops [38, 39]. For all loops from this subclass (which includes all *twn*-loops), whenever we have a runtime bound, the procedure computes finite size bounds. To this end, we over-approximate the effect of a loop by computing closed forms and instantiating the loop counter by the runtime bound. On the other hand, we compute size bounds by considering the local change resulting from a transition. To compute the accumulated change that results from the repeated execution of the transition, we use its runtime bound [14, 41, 45]. This approach is applicable to arbitrary programs and not restricted to loops. Note that KoAT only needs size bounds for complexity, but not for termination analysis.

Control-Flow Refinement. In addition, we integrated *control-flow refinement* [17, 18] into our tool KoAT [25, 42]. The main idea of CFR is to gain information on the values of variables and to sort out certain paths in the program. For example, our adaption of the CFR technique from [18] in KoAT detects that in Fig. 4, after reaching a loop iteration where the **else**-case applies, one never reaches another loop iteration where the **if**-case is used. Therefore, it transforms the integer program corresponding to Fig. 4 into the program corresponding to Fig. 5. While the programs are equivalent, the program in Fig. 5 is easier to analyze, since here the two consecutive loops do not interfere with each other: x and z are constants in its first loop, while y and z are constants in its second loop.

3 Related Work

As mentioned in the introduction, there exist many approaches to analyze complexity of integer programs automatically, e.g., [1–4, 6, 14, 15, 20, 21, 23, 26, 31–33, 35, 47, 53, 55, 56]. Instead of representing integer programs via transitions, there are also techniques based on *cost equation systems*, implemented in the tool CoFloCo [20, 21]. This approach analyzes program parts independently and uses linear invariants to compose the results, i.e., it differs significantly from our approach which can also infer non-linear size bounds. Similarly, in the tool PUBS [1, 3], *cost relations* are analyzed which are a system of recursive equations that capture the cost of the program. There are also numerous approaches for automatic resource analysis of functional programs, often based on amortized analysis (see [33] for an overview). For example, an approach for automatic complexity analysis of OCaml programs is presented in [31, 32], which however has limitations w.r.t. modularity, see [52], and is restricted to polynomial bounds. There are also several approaches based on types, e.g., the resource consumption of Liquid Haskell programs is encoded in a type system in [26], but here bounds are not inferred automatically. Another line of work automatically infers bounds by generating and solving recurrence relations, e.g., [35, 55].

There also exist tools which analyze the runtime complexity of C-code, e.g., Loopus [56] or MaxCore [4] with CoFloCo or PUBS in the backend. For KoAT, we use Clang [16] and llvm2kittel [19] to transform pointer-free C programs into integer transition systems. To prove termination of more general C programs, we developed the framework AProVE (KoAT + LoAT) [43], which participates in the annual *Software Verification Competition* (SV-COMP) [8]. In addition, AProVE also uses KoAT to infer runtime bounds for more general C programs [30]. Moreover, KoAT is used as a backend in the tool Pico for cost analysis of GPU kernels [10].

There is a wealth of work on automated termination analysis of integer programs. Most approaches are based on variants of ranking functions (e.g., [5–7, 29, 54]). Moreover, there are also classes of programs where termination is decidable (e.g., [12, 28, 34, 57, 60]). There exist numerous tools which can automatically prove or disprove termination of integer programs, e.g., Golem [9], iRankFinder [17], LoAT [23], MuVal [59], T2 [13], and VeryMax [11] which participated in last year's TermComp.[2] The tools Golem and LoAT are specialized in *disproving*

[2] In recent years, also several data-driven techniques for termination analysis were developed which performed successfully at competitions like SV-COMP [8], e.g., [48, 51]. However, in general these techniques can yield unsound results.

termination, whereas KoAT can only *prove* termination. Moreover, LoAT infers lower runtime bounds. Thus, by running LoAT in parallel with KoAT one can infer tight asymptotic bounds whenever the bounds of both tools match.

4 Evaluation and Conclusion

We presented the tool KoAT for termination and complexity analysis of integer programs. While the first version of KoAT dates back to [14], this version did not have a specialized mode for termination analysis, it only applied classical ranking functions for the computation of runtime bounds (but no MΦRFs), and it did not use any techniques based on *twn*- or solvable loops. Thus, we re-implemented KoAT completely and integrated the results of our papers [25, 40, 41, 44, 45].

Moreover, our new implementation of KoAT can also analyze *expected* runtimes of *probabilistic* integer programs [42, 49]. To this end, we lifted our modular framework to the probabilistic setting, leading to the notions of *expected* runtime and size bounds. They over-approximate the expected number of evaluation steps with a specific transition and the expected absolute value of a variable after evaluating a specific transition. In general, these expected values may depend on each other, such that it is not always possible to combine already computed expected runtime and size bounds to derive new, improved bounds. In such cases, KoAT falls back to the computation of *non-probabilistic* bounds by over-approximating all probabilistic with non-probabilistic behavior. The obtained non-probabilistic bounds are then integrated into the analysis of the original probabilistic program. Thus, any improvement in the analysis of non-probabilistic programs can also be used when analyzing probabilistic ones.

KoAT is an open-source tool written in OCaml. In the beginning of the analysis, KoAT preprocesses the program, e.g., by extending the guards of transitions with polyhedron invariants inferred by Apron [36]. For all SMT problems (e.g., to infer MΦRFs and to prove termination of *twn*-loops), KoAT calls Z3 [50].

To evaluate the power of KoAT, we used the benchmarks from the TPDB in the categories Complexity_ITS and Termination_ITS for integer transition systems, which are used in the annual TermComp. For the (similar) results of a corresponding evaluation on C programs see [45]. In our evaluation, we compare KoAT with all tools which participated in the respective categories at TermComp. Moreover, we compared KoAT to the original KoAT implementation of [14]. To distinguish this original implementation from our re-implementation, we refer to the tool of [14] as KoAT1 in the following. As mentioned, KoAT1 does not have a specific procedure for termination analysis. For complexity analysis, we evaluated our novel version of KoAT against the tools KoAT1 and CoFloCo. We ran KoAT using its most powerful configuration, in which all our recent improvements are enabled (i.e., MΦRFs, runtime bounds for *twn*-loops, size bounds for solvable loops, and control-flow refinement). Table 1 shows the results of our evaluation on Complexity_ITS, where as in last year's TermComp, we used a timeout of one minute per example. All tools were run inside an Ubuntu Docker container on a machine with an AMD Ryzen 7 3700X octa-core CPU and 8 GB

Table 1: Evaluation for Complexity Analysis (Complexity_ITS, 838 Benchmarks)

	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$< \omega$	AVG$^+$(s)	AVG(s)
KoAT	132	9	261	113	24	548	2.81	7.17
KoAT1	132	0	216	104	14	475	0.67	3.33
CoFloCo	125	0	230	93	9	457	1.50	5.38

Table 2: Evaluation for Termination Analysis (Termination_ITS, 1222 Benchmarks)

Result	T2	iRankFinder	VeryMax	MuVal	KoAT	LoAT	Golem
Terminating	606	632	627	515	**635**	2	0
Non-Terminating	422	385	364	448	0	**522**	350
Total	**1028**	1017	991	963	635	524	350
AVG$^+$(s)	2.03	5.45	6.67	3.96	1.96	1.38	2.42
AVG(s)	7.08	12.99	14.51	15.51	7.94	11.24	9.87

of RAM. The runtime bounds inferred by the tools are compared asymptotically as functions which depend on the largest initial absolute value n of all program variables. So for example, KoAT proved an (at most) linear runtime bound for $132 + 9 + 261 = 402$ benchmarks. So for these examples KoAT can show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(n)$ for all initial states $\sigma_0 \in \Sigma$ where $|\sigma_0|(v) \leq n$ for all $v \in \mathcal{V}$. Overall, this configuration succeeds on 548 examples, i.e., "$< \omega$" is the number of examples where a finite bound on the runtime complexity could be computed by the tool within the time limit. "AVG$^+$(s)" denotes the average runtime of successful runs in seconds, whereas "AVG(s)" is the average runtime of all runs. While KoAT is a bit slower, the table clearly shows that it is considerably more powerful than the two other tools for complexity analysis.

For termination analysis, we compared KoAT against the tools which participated in last year's TermComp: Golem, iRankFinder, LoAT, MuVal, T2, and VeryMax on Termination_ITS, see Table 2. In our experiments, KoAT proved termination for 635 benchmarks in an average runtime of 1.96 seconds. This shows that KoAT is a powerful termination tool even though it is primarily designed for complexity analysis. More precisely, in our experiments, KoAT was the most powerful tool for *proving* termination of programs, whereas LoAT was the most powerful tool for *disproving* termination, which proved non-termination on 522 benchmarks. Overall, T2 solved the most examples with 1028 instances.

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

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

This website also contains details on our input format and a *web interface* to run different configurations of KoAT directly online. The detailed results of our evaluation can be found at https://koat.verify.rwth-aachen.de/evaluation [46].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data Availability Statement. An artifact with KoAT's binary and Docker images in order to reproduce our experiments is available at [37]:

https://doi.org/10.5281/zenodo.18399478

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Proc. SAS '08*. LNCS 5079. 2008, pp. 221–237. DOI: 10.1007/978-3-540-69166-2_15.

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

[3] E. Albert, S. Genaim, and A. N. Masud. "On the Inference of Resource Usage Upper and Lower Bounds". In: *ACM Transactions on Computational Logic* 14.3 (2013). DOI: 10.1145/2499937.2499943.

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

[5] A. M. Ben-Amram and S. Genaim. "Ranking Functions for Linear-Constraint Loops". In: *Journal of the ACM* 61.4 (2014), 26:1–26:55. DOI: 10.1145/2629488.

[6] A. M. Ben-Amram and S. Genaim. "On Multiphase-Linear Ranking Functions". In: *Proc. CAV '17*. LNCS 10427. 2017, pp. 601–620. DOI: 10.1007/978-3-319-63390-9_32.

[7] A. M. Ben-Amram, J. J. Doménech, and S. Genaim. "Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets". In: *Proc. SAS '19*. LNCS 11822. 2019, pp. 459–480. DOI: 10.1007/978-3-030-32304-2_22.

[8] D. Beyer and J. Strejček. "Improvements in Software Verification and Witness Validation: SV-COMP 2025". In: *Proc. TACAS '25*. LNCS 15698. Website of SV-COMP: https://sv-comp.sosy-lab.org. 2025, pp. 151–186. DOI: 10.1007/978-3-031-90660-2_9.

[9] M. Blicha, K. Britikov, and N. Sharygina. "Golem: A Flexible and Efficient Solver for Constrained Horn Clauses". In: *Formal Methods in System Design* 67 (2025), pp. 143–160. DOI: 10.1007/s10703-025-00470-9.

[10] G. Blike, H. Zicarelli, U. Sathiyamoorthy, J. Lange, and T. Cogumbreiro. "A Modular Static Cost Analysis for GPU Warp-Level Parallelism". In: *Proceedings of the ACM on Programming Languages* 10.POPL (2026), pp. 1471–1499. DOI: 10.1145/3776693.

[11] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "Proving Termination Through Conditional Termination". In: *Proc. TACAS '17*. LNCS 10205. 2017, pp. 99–117. DOI: 10.1007/978-3-662-54577-5_6.

[12] M. Braverman. "Termination of Integer Linear Programs". In: *Proc. CAV '06*. LNCS 4144. 2006, pp. 372–385. DOI: 10.1007/11817963_34.

[13] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. "T2: Temporal Property Verification". In: *Proc. TACAS '16*. LNCS 9636. 2016, pp. 387–393. DOI: 10.1007/978-3-662-49674-9_22.

[14] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. "Analyzing Runtime and Size Complexity of Integer Programs". In: *ACM Transactions on Programming Languages and Systems* 38 (2016), pp. 1–50. DOI: 10.1145/2866575.

[15] Q. Carbonneaux, J. Hoffmann, and Z. Shao. "Compositional Certified Resource Bounds". In: *Proc. PLDI '15*. 2015, pp. 467–478. DOI: 10.1145/2737924.2737955.

[16] Clang Compiler. URL: https://clang.llvm.org/.

[17] J. J. Doménech and S. Genaim. "iRankFinder". In: *Proc. WST '18*. https://wst2018.webs.upv.es/wst2018proceedings.pdf. 2018, p. 83.

[18] 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 and Practice of Logic Programming* 19.5-6 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

[19] S. Falke, D. Kapur, and C. Sinz. "Termination Analysis of C Programs Using Compiler Intermediate Languages". In: *Proc. RTA '11*. LIPIcs 10. 2011, pp. 41–50. DOI: 10.4230/LIPIcs.RTA.2011.41.

[20] A. Flores-Montoya and R. Hähnle. "Resource Analysis of Complex Programs with Cost Equations". In: *Proc. APLAS '14*. LNCS 8858. 2014, pp. 275–295. DOI: 10.1007/978-3-319-12736-1_15.

[21] A. Flores-Montoya. "Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations". In: *Proc. FM '16*. LNCS 9995. 2016, pp. 254–273. DOI: 10.1007/978-3-319-48989-6_16.

[22] F. Frohn and C. Fuhs. "A Calculus for Modular Loop Acceleration and Non-Termination Proofs". In: *International Journal on Software Tools for Technology Transfer* 24.5 (2022), pp. 691–715. DOI: 10.1007/S10009-022-00670-2.

[23] F. Frohn and J. Giesl. "Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)". In: *Proc. IJCAR '22*. LNCS 13385. 2022, pp. 712–722. DOI: 10.1007/978-3-031-10769-6_41.

[24] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *Proc. TACAS '19*. LNCS 11429. Website of TermComp: https://termination-portal.org/wiki/Termination_Competition. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[25] 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.

[26] M. A. T. Handley, N. Vazou, and G. Hutton. "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020). DOI: 10.1145/3371092.

[27] M. Hark, F. Frohn, and J. Giesl. "Polynomial Loops: Beyond Termination". In: *Proc. LPAR '20*. EPiC 73. 2020, pp. 279–297. DOI: 10.29007/nxv1.

[28] M. Hark, F. Frohn, and J. Giesl. "Termination of Triangular Polynomial Loops". In: *Formal Methods in System Design* 65.1 (2025), pp. 70–132. DOI: 10.1007/s10703-023-00440-z.

[29] M. Heizmann and J. Leike. "Ranking Templates for Linear Loops". In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015.

[30] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. "Termination and Complexity Analysis for Programs with Bitvector Arithmetic by Symbolic Execution". In: *Journal of Logic and Algebraic Methods in Programming* 97 (2018), pp. 105–130. DOI: 10.1016/J.JLAMP.2018.02.004.

[31] J. Hoffmann, K. Aehlig, and M. Hofmann. "Multivariate Amortized Resource Analysis". In: *ACM Transactions on Programming Languages and Systems* 34.3 (2012). DOI: 10.1145/2362389.2362393.

[32] J. Hoffmann, A. Das, and S.-C. Weng. "Towards Automatic Resource Bound Analysis for OCaml". In: *Proc. POPL '17*. 2017, pp. 359–373. DOI: 10.1145/3009837.3009842.

[33] J. Hoffmann and S. Jost. "Two Decades of Automatic Amortized Resource Analysis". In: *Mathematical Structures in Computer Science* 32.6 (2022), pp. 729–759. DOI: 10.1017/S0960129521000487.

[34] M. Hosseini, J. Ouaknine, and J. Worrell. "Termination of Linear Loops over the Integers". In: *Proc. ICALP '19*. LIPIcs 132. 2019. DOI: 10.4230/LIPIcs.ICALP.2019.118.

[35] D. Ishimwe, K. Nguyen, and T. Nguyen. "Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021). DOI: 10.1145/3485515.

[36] B. Jeannet and A. Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Proc. CAV '09*. LNCS 5643. 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[37] KoAT*: Automatic Complexity and Termination Analysis of Integer Programs*. Zenodo. 2026. DOI: 10.5281/zenodo.18399478.

[38] L. Kovács, N. Popov, and T. Jebelean. "Combining Logic and Algebraic Techniques for Program Verification in Theorema". In: *Proc. ISoLA '06*. 2006, pp. 67–74. DOI: 10.1109/ISoLA.2006.46.

[39] L. Kovács. "Reasoning Algebraically About P-Solvable Loops". In: *Proc. TACAS '08*. LNCS 4963. 2008, pp. 249–264. DOI: 10.1007/978-3-540-78800-3_18.

[40] N. Lommen, F. Meyer, and J. Giesl. "Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops". In: *Proc. IJCAR '22*. LNCS 13385. 2022, pp. 734–754. DOI: 10.1007/978-3-031-10769-6_43.

[41] N. Lommen and J. Giesl. "Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs". In: *Proc. FroCoS '23*. LNCS 14279. 2023, pp. 3–22. DOI: 10.1007/978-3-031-43369-6_1.

[42] N. Lommen, É. Meyer, and J. Giesl. "Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper)". In: *Proc. IJCAR '24*. LNCS 14739. 2024, pp. 233–243. DOI: 10.1007/978-3-031-63498-7_14.

[43] N. Lommen and J. Giesl. "AProVE (KoAT + LoAT) (Competition Contribution)". In: *Proc. TACAS '25*. LNCS 15698. 2025, pp. 205–211. DOI: 10.1007/978-3-031-90660-2_13.

[44] N. Lommen and J. Giesl. "Modular Automatic Complexity Analysis of Recursive Integer Programs". In: *Proc. ESOP '26*. LNCS 16502. Full version available in *Corr* abs/2512.18851, https://doi.org/10.48550/arXiv.2512.18851. 2026, pp. 1–31. DOI: 10.1007/978-3-032-22723-2_1.

[45] N. Lommen, É. Meyer, and J. Giesl. "Targeting Completeness: Automated Complexity Analysis of Integer Programs". In: *Journal of Automated Reasoning* 70:6 (2026). DOI: 10.1007/s10817-026-09751-2.

[46] N. Lommen, É. Meyer, and J. Giesl. *Experiments and Details for "*KoAT*: Automatic Complexity and Termination Analysis of Integer Programs"*. 2026. URL: https://koat.verify.rwth-aachen.de/evaluation.

[47] P. López-García, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. "Interval-Based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption". In: *Theory and Practice of Logic Programming* 18.2 (2018), pp. 167–223. DOI: 10.1017/S1471068418000042.

[48] R. Metta, P. Yeduru, H. Karmarkar, and R. K. Medicherla. "VeriFuzz 1.4: Checking for (Non-)Termination (Competition Contribution)". In: *Proc. TACAS '23*. LNCS 13994. 2023, pp. 594–599. DOI: 10.1007/978-3-031-30820-8_42.

[49] F. Meyer, M. Hark, and J. Giesl. "Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes". In: *Proc. TACAS '21*. LNCS 12651. 2021, pp. 250–269. DOI: 10.1007/978-3-030-72016-2_14.

[50] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. TACAS '08*. LNCS 4963. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[51] D. Mukhopadhyay, R. Metta, H. Karmarkar, and K. Madhukar. "PROTON 2.1: Synthesizing Ranking Functions via Fine-Tuned Locally Hosted LLM (Competition Contribution)". In: *Proc. TACAS '25*. LNCS 15698. 2025, pp. 242–247. DOI: 10.1007/978-3-031-90660-2_19.

[52] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. "Complexity Analysis for Term Rewriting by Integer Transition Systems". In: *Proc. FroCoS '17*. LNCS 10483. 2017, pp. 132–150. DOI: 10.1007/978-3-319-66167-4_8.

[53] L. Pham, F. A. Saad, and J. Hoffmann. "Robust Resource Bounds with Static Analysis and Bayesian Inference". In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024). DOI: 10.1145/3656380.

[54] A. Podelski and A. Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Proc. VMCAI '04*. LNCS 2937. 2004, pp. 239–251. DOI: 10.1007/978-3-540-24622-0_20.

[55] L. Rustenholz, M. Klemen, M. Á. Carreira-Perpiñán, and P. López-García. "A Machine Learning-Based Approach for Solving Recurrence Relations and its use in Cost Analysis of Logic Programs". In: *Theory and Practice of Logic Programming* 24.6 (2024), pp. 1163–1207. DOI: 10.1017/S1471068424000413.

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

[57] A. Tiwari. "Termination of Linear Programs". In: *Proc. CAV '04*. LNCS 3114. 2004, pp. 70–82. DOI: 10.1007/978-3-540-27813-9_6.

[58] TPDB (Termination Problems Data Base). URL: https://github.com/TermCOMP/TPDB-ARI.

[59] H. Unno, T. Terauchi, Y. Gu, and E. Koskinen. "Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification". In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 2111–2140. DOI: 10.1145/3571265.

[60] M. Xu and Z.-B. Li. "Symbolic Termination Analysis of Solvable Loops". In: *Journal of Symbolic Computation* 50 (2013), pp. 28–49. DOI: 10.1016/j.jsc.2012.05.005.