Final Report: Automated Termination and Complexity Analysis of Imperative Programs

1 General Information

DFG reference number:	GI 274/6-2
Project number:	235950644
Project title:	Automated Termination and Complexity Analysis of Imperative Programs
Name of the applicant:	Prof. Dr. Jürgen Giesl
Official address:	LuFG Programming Languages and Verification, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
Reporting period:	1. 4. 2021 - 31. 3. 2024 (GI 274/6-2) 1. 3. 2014 - 28. 2. 2018 (GI 274/6-1)

2 Summary

Zusammenfassung:

Die *Terminierungs-* und *Komplexitätsanalyse* imperativer Programme ist ein zentraler Punkt der Programmverifikation. Hierzu wurde folgender Ansatz verwendet:

Das *Front-End* erzeugt aus dem Programm einen *symbolischen Auswertungsgraph*, der alle Programmläufe repräsentiert. Daraus wird ein Programm in einer einfachen Sprache generiert, z.B. ein Termersetzungssystem (TRS) oder ein Integer Transitionssystem (ITS), dessen Terminierung oder Komplexität (im *Back-End*) untersucht wird. Folgende Resultate wurden erzielt:

(A) Terminierungsanalyse
Basierend auf unseren Vorarbeiten zur Terminierung von Java wurde ein Verfahren zur Terminierungsanalyse von C Programmen entwickelt, um sowohl maschinennahe Speicheroperationen als auch Konzepte wie Datenstrukturen, Rekursion und Bitvektor-Zahlen zu behandeln.

Während Terminierung im Allgemeinen unentscheidbar ist, wurden Klassen von Programmen identifiziert, bei denen Fragen der Terminierung oder Komplexität entscheidbar sind und entsprechende vollständige Techniken vorgestellt.

Zur Verbesserung des *Back-Ends* wurden mehrere Beiträge zur Terminierung von TRSen entwickelt. Dies betrifft neue Ansätze zur *relativen* Terminierung und zur "fast sicheren" Terminierung von *probabilistischen* TRSen sowie den Einsatz der Terminierungsanalyse von TRSen zur Minimierung von Beweisen in Beschreibungslogiken.

(B) Komplexitätsanalyse (Obere Schranken)
Terminierungs-Techniken lassen sich modifizieren, um damit *obere Schranken* für die Laufzeit von Programmen zu berechnen. Dies betrifft sowohl die Front-Ends für Java und C also auch die Back-Ends für TRSe und ITSe. Insbesondere wurde ein modulares Verfahren zur Komplexitätsanalyse von ITSen entwickelt, das Schranken für die *Laufzeit* und die *Werte* der Programmvariablen für einzelne Teilprogramme berechnet. Um seine Mächtigkeit zu erhöhen, wurden vollständige Verfahren für Teilprogramme mit berechenbaren Schranken (wie in **(A)**) integriert. Der Ansatz wurde auch auf die Berechnung *erwarteter* Laufzeitschranken für *probabilistische* ITSe erweitert.

(C) Komplexitätsanalyse (Untere Schranken)
Zur Fehlersuche wurden Techniken zur Berechnung *unterer Laufzeitschranken* für Back-End Programme entwickelt. Für ITSe beruhen diese auf einer *Loop Acceleration*, die Schleifen in nichtdeterministische Instruktionen übersetzt. Hierzu wurden neue Kalküle vorgestellt, die Loop Acceleration mit der Suche nach Gegenbeispielen verbinden. Diese lassen sich auch für Nicht-Terminierungsbeweise und zur Erfüllbarkeitsanalyse von *Constraint Horn Clauses* verwenden (um Sicherheitseigenschaften von Programmen zu untersuchen). Da Loop Acceleration oft exponentielle Ausdrücke erzeugt, wurde auch eine entsprechende Erweiterung von SMT Solvern entwickelt.

(D) Implementierung
Alle Resultate wurden in unseren Tools AProVE (für den Gesamt-Ansatz), KoAT und LoAT (für obere und untere Schranken bei ITSen) und SwInE (für SMT modulo exponentieller Arithmetik) implementiert und in umfangreichen Experimenten und Wettbewerben evaluiert.

Summary:

Our goal was to analyze *termination* and *complexity* of imperative programs, which is crucial for software verification. We used the following methodology:

In the *front-end*, a finite *symbolic execution graph* is generated from the program, which represents all possible program executions. Then, a program in a simple language, e.g., a term rewrite system (TRS) or an integer transition system (ITS) is synthesized from the graph. Finally, termination or complexity of this simple program is analyzed in the *back-end* of our approach. We obtained the following contributions:

(A) Termination Analysis
Based on our previous work on termination of Java, we developed an approach for automatic termination analysis of C programs, which can analyze programs with low-level memory operations and features like dynamic data structures, recursion, or bitvector integers.

While termination is undecidable in general, we also developed results on classes of programs where questions of termination or complexity are decidable. So for sub-programs from such a class, one can apply complete techniques.

To improve the power of the *back-end*, we obtained several new results on termination of TRSs. We developed novel frameworks for *relative* termination and for almost-sure termination of *probabilistic* TRSs, and results on using termination analysis of TRSs to minimize description logic proofs.

(B) Complexity Analysis (Upper Bounds)
We adapted techniques for proving termination to infer *upper bounds* on the runtime of programs. This concerns both the front-ends for Java and C, and the back-ends for TRSs and ITSs. In particular, we developed a modular approach for complexity analysis of ITSs which alternates between computing *runtime* and *size bounds* (on the values of program variables) for program parts. To increase its power, we also integrated complete techniques (as in **(A)**) for sub-programs where runtime or size bounds are computable. Moreover, we adapted the approach in order to compute upper bounds on the *expected* runtimes of *probabilistic* ITSs.

(C) Complexity Analysis (Lower Bounds)
To detect bugs, we also developed techniques to compute *lower runtime bounds* for back-end programs. For ITSs, these techniques rely on *accelerating loops* into non-deterministic straight-line code. We developed new calculi to combine loop acceleration with the search for counterexamples in order to use loop acceleration not just for lower bounds, but also to prove non-termination of ITSs or (un)satisfiability of *Constraint Horn Clauses* (which are used to analyze program safety). Since loop acceleration often generates exponential expressions, to check reachability in accelerated programs, we also extended SMT solvers to exponential integer arithmetic.

(D) Implementation
We implemented all our contributions in the tools AProVE (for the overall approach), KoAT and LoAT (for upper and lower bounds of ITSs), and SwInE (for SMT modulo exponential arithmetic). Their power was evaluated in extensive experiments and annual competitions.

3 Progress Report

In the following, we describe the results achieved in the project. For further details we refer to the papers [P1] - [P20] which resulted from the project during the period 2021 - 2024 as well as to the

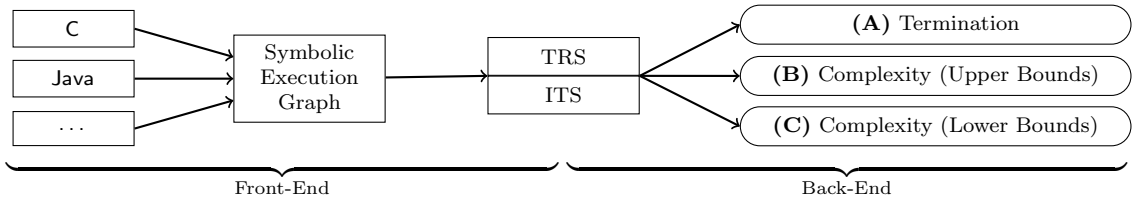

Figure 1: Structure of our Approach

papers [7, 13–16, 18, 20, 21, 25, 26, 29, 32, 33] from the first funding period (2014 - 2018), see Sect. 4. The results and publications from this first period were already reported in the renewal proposal for the second phase (submitted to the DFG in 2019), but we recapitulate them here as well.

In the project, we developed a powerful automated approach to analyze termination and complexity of (sequential) imperative programs [21], as illustrated in Fig. 1. First, the program is transformed into a *symbolic execution graph* (SEG) which is a finite over-approximation of all program runs. Afterwards, a back-end program (a *term rewrite system* (TRS) or an *integer transition system* (ITS)) is automatically generated from this graph, whose termination (or complexity) is analyzed subsequently. In this way, one only has to develop techniques to analyze simple back-end languages like TRSs or ITSs instead of full programming languages. Moreover, the same back-end can be used for the analysis of many different source languages.

(A) Termination Analysis

We developed the following new contributions on automated termination analysis:

(A.1) Termination Analysis for C [P2, P8, P20]

In addition to the "high-level" language Java [5, 6, 30], in [32, 33] we adapted our approach to analyze programs in the "machine-oriented" language C where the program controls the memory management and termination can depend on heap manipulations based on explicit pointer arithmetic. To avoid handling the intricacies of C, we analyze the intermediate representation of the LLVM framework [28] and developed a suitable domain for the abstract states of the SEG that can track allocated memory in detail. Our domain is based on separation logic [31], but standard SMT solving is sufficient for all reasoning required during our symbolic execution. To prove termination of C programs with low-level memory access, our approach also verifies memory safety.

We started handling programs with integers, pointers, and arrays. Thus, when analyzing C programs, we generate ITSs (instead of TRSs) from the SEG. Since in most languages, integers are implemented as fixed-width bitvectors, in [26] we adapted our approach to take overflows into account. Moreover, we developed the following improvements to increase the power of our approach:

***Improving Modularity and Handling Recursion* [P20].** In [25] and [P20] we showed how to combine the byte-accurate representation of the memory with the modular analysis of (possibly recursive) functions. To this end, we developed a new technique to abstract the call stack in the abstract states of the SEG, which ensures that the graph construction also terminates for functions with recursive calls. Moreover, our novel abstraction of call stacks allows us to construct separate SEGs for every function and to re-use previous analyses of (possibly recursive) auxiliary functions in a modular way. In this way, every function is analyzed individually and its termination does not have to be re-proved anymore when it is called by another function.

***Generating Witnesses for Non-Termination* [P2].** Sometimes, the SEG does not contain over-approximating steps but it models the program precisely. Then, non-termination of the ITS resulting from a cycle of the graph together with a path from the root of the graph to the cycle implies non-termination of the program [25]. We refer to [17] and [P4, P6, P7, P10] in Part **(C)** for our new results and our corresponding tool LoAT to prove non-termination of ITSs automatically.

In [P2], we showed how to generate *non-termination witnesses* to *certify* the non-termination proofs of C programs. To this end, the variable assignment obtained from the non-terminating cycle is used to step through the SEG in order to obtain a concrete execution path which witnesses

non-termination. From this path through the SEG, AProVE extracts the LLVM program positions, which are then matched to the lines in the original C program for certification. In addition, certification has also been extended to the front-end such that the translation of LLVM to ITSs via the SEG can be certified as well [24].

***Combining Byte-Precise Memory Modeling with Data Structures* [P8].** In [P8] we introduced a novel technique to combine our byte-precise modeling of the memory with the analysis of C programs on lists. To this end, we developed a general approach to infer *list invariants* via symbolic execution, which allow us to abstract from detailed information about lists (e.g., about their intermediate elements) such that abstract states with "similar" lists can be merged and generalized during the symbolic execution in order to obtain finite SEGs. At the same time, list invariants contain enough information about the lists such that memory safety and termination can usually still be proved. For example, a list invariant represents the length, the start address, and knowledge about some of the contents of the list, as well as information on the memory arrangement of the list fields, which is needed for programs that access fields via pointer arithmetic.

We implemented our approach for termination analysis of C programs in our tool AProVE [21], and participated very successfully in the annual *International Competition on Software Verification (SV-COMP[1])* at the *TACAS* conference and the annual *International Termination and Complexity Competition (TermComp[2] [20, 22])*, both of which feature categories for termination of C programs since 2014. The performance of AProVE at these competitions clearly demonstrates the power and usefulness of our improvements in [P20, P2, P8].

(A.2) Classes with Complete Termination Analysis Techniques [P12]

While termination is undecidable, we investigated sub-classes of programs where questions of termination or complexity become decidable. Thus, in [P12] we studied *complete* approaches for classes of polynomial loops. As shown in [P5, P11] in Part **(B.1)**, these results can also be integrated into (incomplete) approaches for general integer programs, which improves their power significantly.

In [P12] we considered so-called *twn*-loops, where the update is "triangular" and "weakly non-linear". The guard of such a loop is a Boolean combination of (possibly non-linear) polynomial inequations, and the body is an assignment of the form $(x_1, \ldots, x_d) \leftarrow (c_1 \cdot x_1 + p_1, \ldots, c_d \cdot x_d + p_d)$. Here, each x_i is a variable, c_i is a number, and p_i is a (possibly non-linear) polynomial over the variables $x_{i+1}, \ldots, x_d$. So an example for a *twn*-loop is **while** $(x_1^2 + x_3^5 < x_2 \wedge x_1 \neq 0)$ **do** $(x_1, x_2, x_3) \leftarrow (-2 \cdot x_1,\ 3 \cdot x_2 - 2 \cdot x_3^3,\ x_3)$.

For such loops, by handling one variable after the other, one can compute a *closed form* which corresponds to applying the loop's update n times (for a fresh variable n). Using these closed forms, termination can be reduced to (in)validity of an existential formula. Our decidability results on termination then follow from existing results on the decidability of certain arithmetic theories. Moreover, we obtained the first computability results on the *runtime complexity* of such loops and showed that for *twn*-loops over the integers, one can always compute a polynomial upper bound on the runtime for all those inputs where the loop terminates [23]. We also analyzed the complexity of (dis)proving termination via our approach and identified a class of linear loops where our approach is efficient.

(A.3) Termination Analysis for Variants of Term Rewriting [P9, P13, P14, P16, P18]

As shown in Fig. 1, we transform programs in complex languages to programs in a simple back-end language, like TRSs or ITSs. TRSs can represent user-defined tree-shaped data structures in a precise way, whereas data objects have to be abstracted to numbers if one uses ITSs in the back-end. For that reason, we use a back-end language which combines TRSs and ITSs [21] for termination analysis of Java [5, 6, 30]. To extend the applicability of TRS-based back-ends, we also worked on techniques for termination analysis for different variants of TRSs.

[1] `https://sv-comp.sosy-lab.org/`

[2] `https://termination-portal.org/wiki/Termination_Competition`

***Almost-Sure Innermost Termination of Probabilistic TRSs* [P9, P13, P14].** *Probabilistic* programs describe randomized algorithms and probability distributions, with applications in many areas. *Probabilistic TRSs* (PTRSs) [4] allow to use TRSs also for such programs. While TRSs have rules of the form $\mathsf{f}(x) \to x$ or $\mathsf{f}(x) \to \mathsf{f}(\mathsf{f}(x))$, PTRSs have rules like $\mathsf{f}(x) \to \{1/2 : x, 1/2 : \mathsf{f}(\mathsf{f}(x))\}$ with multi-distributions on the right-hand side. This rule corresponds to a symmetric random walk where the number of f-symbols is increased or decreased by 1, both with probability 1/2. This PTRS is *almost-surely terminating* (AST), i.e., the probability for termination is 1.

While numerous techniques exist to prove AST of imperative programs on numbers [8,27], there are only few automatic approaches for programs with complex non-tail recursive structure and recursive data structures. The only previous approach for automatic termination analysis of PTRSs [2] adapted orderings based on interpretations to PTRSs. However, already for non-probabilistic TRSs such a *direct* application of orderings is limited in power. For a powerful approach, one should combine such orderings in a modular way, as in the *dependency pair* (DP) framework for (non-probabilistic) TRSs [1, 19], which is used in essentially all current termination tools for TRSs.

Therefore, in [P9, P14] we presented the first adaption of the DP framework to the probabilistic setting in order to prove AST for rewrite sequences which follow the innermost evaluation strategy. While the approach of [P9] was based on an incomplete DP criterion, we increased its power and improved it to a complete criterion using so-called *annotated* dependency pairs (ADPs) in [P14].

For non-probabilistic TRSs, several criteria exist when innermost termination implies full termination. In [P13], we developed the first such criteria for the probabilistic setting and in this way, the ADP framework for innermost AST can now also prove AST of full rewriting.

The evaluation of our implementation in our tool AProVE showed that the power of the DP framework can now indeed also be used for probabilistic TRSs. So in the future, this allows the application of such TRSs in the back-end when analyzing probabilistic programs.

***Applications of Termination Analysis Techniques for TRSs* [P16, P18].** To allow the use of TRSs as a back-end for further applications, in [P16] we showed that the concept of annotated DPs from [P14] can be adapted in order to analyze *relative termination*. Here, infinite rewrite sequences are allowed, but one wants to prove that a certain subset of the rewrite rules cannot be used infinitely often. This answers a long-standing open problem (# 106 in the open problem list of [10]) and allows to solve relative termination problems in many applications.

Moreover, we studied the application of TRSs $\mathcal{R}$ to make proofs for description logic inferences smaller. In [P18], we showed that the complexity of the corresponding decision problem depends on how termination of $\mathcal{R}$ can be established and gave tight complexity bounds for several classes of termination orderings.

(B) Complexity Analysis (Upper Bounds)

In practice, termination is often not sufficient, but one also has to ensure that algorithms terminate in *reasonable* (e.g., polynomial) *time*. To this end, one would like to find upper bounds on the worst-case *runtime complexity* of programs automatically.

To adapt the front-end of our approach (see Fig. 1) from termination to complexity analysis, for Java programs, we defined a novel transformation of symbolic execution graphs to ITSs in [15], which over-approximates how the sizes of objects change due to sharing and aliasing on the heap. Thus, when analyzing the complexity of the resulting ITS in the back-end, the obtained upper bound is also an upper bound for the original Java program. Moreover, building upon our adaption of termination analysis to bitvector programs with potential overflows, we showed in [26] that our approach can also be used to analyze the runtime complexity of C bitvector programs.

In the back-end, we improved techniques for complexity analysis of both TRSs and ITSs. For TRSs, we showed that it is semi-decidable if the runtime complexity of a TRS is *constant* [16], and similar to the work in [P13] we developed a sufficient criterion which ensures that the runtime complexity of full rewriting for a TRS coincides with its innermost runtime complexity [14]. Moreover, we developed a transformation from TRSs to a generalized notion of ITSs such that approaches for

complexity analysis of ITSs can also be applied to analyze TRSs [29]. To infer upper bounds on the runtime of integer programs, we obtained the following results:

(B.1) Complexity Analysis for Integer Transition Systems [P3, P5, P11]

In [7] we developed a novel modular approach for complexity analysis of ITSs which *alternates* between finding *runtime bounds* (by generating suitable ranking functions) and finding *size bounds* on the absolute values of program variables. In this way, one only has to consider small parts of the ITS in each step, and the process is repeated until all loops and variables have been handled. Extensive experiments with our implementation in a new tool KoAT and its success at the *TermComp* competition show its performance and power, also in comparison with other tools.

***Improving Complexity Analysis for ITSs* [P3].** In [P3], we integrated recent techniques to improve automated termination analysis of integer programs into our approach for the inference of runtime bounds from [7]. More precisely, we integrated *multiphase-linear ranking functions* [3] to handle loops whose behavior corresponds to several different subsequent phases, and *control-flow refinement* [11] to gain more information on the values of variables by sorting out certain infeasible paths in the program. As shown by our experimental evaluation, due to these new improvements, the resulting implementation in our new version of KoAT outperforms the other existing tools for complexity analysis of integer programs.

***Integrating Complete Techniques* [P5, P11].** As discussed in **(A.2)**, we developed several results on deciding termination and computing runtime bounds for *twn*-loops [P12]. To use such complete techniques for sub-classes of programs within our incomplete approach for complexity analysis of full integer programs from [7] and [P3], in [P5] we developed a technique to compute *local runtime bounds* for sub-programs which can be transformed into *twn*-loops. Similarly, in [P11] we presented a new procedure to infer *size bounds* which is complete for a sub-class of loops. Size bounds are then used to lift local runtime bounds to global runtime bounds for the whole program.

Due to the integration of our complete techniques, our tool KoAT can now analyze complexity of programs (possibly with non-linear arithmetic) where all other state-of-the-art tools fail. Moreover, KoAT now also has become one of the most powerful tools for termination analysis of ITSs. This demonstrates that results on sub-classes of programs with computable complexity bounds are not only theoretically interesting, but they have an important practical value.

(B.2) Expected Runtimes for Probabilistic Integer Transition Systems [P1, P15]

In **(A.3)**, we considered an adaption of the TRS back-end language to the probabilistic setting and developed techniques to analyze AST of PTRSs [P9, P13, P14]. To analyze probabilistic programs on integers as well, in [P1] we extended our modular approach for complexity analysis of ITSs from [7] to *probabilistic* ITSs. The resulting approach again computes bounds on the runtimes of program parts and on the sizes of their variables in an alternating way, but now it infers upper bounds on the *expected* runtimes of probabilistic ITSs. When computing expected runtime or size bounds for new program parts, the main difficulty is to determine when it is sound to use *expected* bounds on previous program parts and when one has to use *non-probabilistic* bounds instead. Moreover, in [P15] we also adapted our improvement via control-flow refinement from [P3] to the probabilistic setting. For an experimental evaluation, we again implemented our approach in our tool KoAT, and demonstrated its power compared to other related tools.

(C) Complexity Analysis (Lower Bounds)

When deducing upper runtime bounds as in **(B)**, one usually considers "worst-case complexity", i.e., for any possible size m, one analyzes the length of the longest execution starting from an input of size m. In many cases it is also important to find *lower bounds* for this notion of complexity (i.e., we are interested in "worst-case lower bounds"). While upper bounds for runtime complexity help to prove the absence of bugs that worsen the performance of programs, lower bounds can be used to find such bugs. In the project, we developed techniques to infer lower runtime bounds for both back-end languages (TRSs and ITSs).

To infer lower bounds for the runtime complexity of TRSs, in [13] we showed how to detect *decreasing loops* (which result in families of rewrite sequences with linear, exponential, or infinite runtime) and an *induction technique* which can also deduce non-linear polynomial lower bounds.

For ITSs, in [18] we developed a framework for under-approximating program simplification whose core is a method for *loop acceleration*. From the simplified programs, our technique then deduces asymptotic lower bounds. Moreover, we also developed an approach to use our loop acceleration technique to prove *non-termination* of ITSs [17]. We implemented our techniques in a new tool LoAT and showed that it infers non-trivial lower bounds for a large number of ITSs and that it is competitive with the state-of-the-art in proving non-termination of integer programs.

To improve our loop acceleration techniques further and to increase their applicability, in the project we developed the following main new contributions:

(C.1) Improving Loop Acceleration [P4, P6]

In [P4] we generalized our approach by introducing a calculus for loop acceleration which combines different acceleration techniques in a modular way. The idea of loop acceleration is to transform loops like **while** $(x > 0)$ **do** $(x, y) \leftarrow (x - 1,\ 3 \cdot y)$ into non-deterministic straight-line code like $(x, y) \leftarrow (x - n,\ 3^n \cdot y)$. So the transformed program updates the program variables in a single step to the same values as n iterations of the original loop. To achieve that, similar to our approach in **(A.2)**, the loop body is transformed into a *closed form*, which is parameterized in n. To be sound, constraints on n (e.g., $x - n + 1 > 0$) are generated to ensure that the original loop allows for at least n iterations. In this way, one can compute symbolic under-approximations of programs, i.e., every execution path in the resulting transformed program corresponds to a path in the original program, but not necessarily vice versa. In [P6], we integrated the loop acceleration calculus into the framework of our tool LoAT in order to apply it to general ITSs instead of just single loops.

(C.2) Loop Acceleration for (Un)satisfiability and Non-Termination [P7, P10, P19]

Loop acceleration can not only be applied for static program analyses like the inference of complexity bounds or proving non-termination, but in [P7] we developed a novel calculus for *Accelerated Driven Clause Learning* (ADCL) in order to (dis-)prove satisfiability of *Constrained Horn Clauses (CHCs)*. CHCs are often used in automated program verification to analyze safety of programs. Here, acceleration is applied to learn new clauses and thus, to avoid repeated derivations in resolution proofs. Therefore, ADCL is particularly useful for finding long counterexamples that are challenging for other techniques. In [P10] we also developed a variant of ADCL for proving non-termination of ITSs where acceleration is used to quickly check reachability of non-terminating loops.

While ADCL performs a depth-first search for counterexamples, in [P19] we presented a corresponding algorithm for *Accelerated Bounded Model Checking* (ABMC) that performs a breadth-first search by integrating acceleration techniques into SMT-based bounded model checking. In this way, shortcuts that compress many execution steps into a single one are computed on the fly in order to detect deep counterexamples within only few proof steps.

Our implementation of ADCL and ABMC in LoAT and an experimental comparison with other leading solvers show that LoAT is very competitive for proving unsatisfiability of CHCs and it is now the leading tool for disproving non-termination of ITSs (which makes it particularly useful as a back-end for termination analysis, see [P2] in **(A.1)**).

(C.3) SMT Solving for Checking Reachability via Loop Acceleration [P17]

When computing closed forms of loops (e.g., for *twn*-loops as in **(A.2)** or via loop acceleration as in **(C.1)** and **(C.2)**), one often obtains integer expressions that contain exponential terms like 3^n. However, most SMT solvers cannot handle such expressions and thus, our approaches based on loop acceleration may fail to analyze reachability.

Therefore, in [P17] we adapted incremental linearization [9] to polynomial integer arithmetic with exponential functions. The idea is to regard exponentiation as an uninterpreted function and to compute suitable lemmas that eliminate models from the search space if they violate the semantics of exponentiation. We implemented our approach in a novel tool SwInE that can be coupled with back-

end SMT solvers for non-linear integer arithmetic, and showed that it is highly effective in practice.

(D) Implementation in the Tools AProVE, KoAT, LoAT, and SwInE

The overall approach of Fig. 1 with the contributions of Part **(A)** is implemented in our tool AProVE, with the sub-tools KoAT and LoAT for the inference of upper and lower complexity bounds for ITSs (implementing the contributions of Parts **(B)** and **(C)**, respectively). Moreover, our novel tool SwInE is used for SMT solving modulo exponential integer arithmetic, see **(C.3)**.

To make our results applicable in practice, our tools and our experimental data are available via public web sites, *GitHub*, and/or *Zenodo*. Moreover, AProVE and KoAT can also be accessed via a web interface and AProVE can be used as a plug-in of the well-known Eclipse development platform [12], see [21]. For more information, we refer to:

- `https://aprove.informatik.rwth-aachen.de/`
- `https://koat.verify.rwth-aachen.de/`
- `https://loat-developers.github.io/LoAT/`
- `https://ffrohn.github.io/swine/`

We evaluated the power of all our contributions by comparing our implementation with related tools in extensive experiments on large collections of examples, as well as by successfully participating in the respective categories of the *TermComp*, *SV-COMP*, and *CHC-COMP*[3] competitions.

Bibliography

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236:133–178, 2000.

[2] M. Avanzini, U. Dal Lago, and A. Yamada. On probabilistic term rewriting. *Sci. Comput. Program.*, 185, 2020.

[3] A. M. Ben-Amram and S. Genaim. On multiphase-linear ranking functions. In *Proc. CAV '17*, LNCS 10427, pages 601–620, 2017.

[4] O. Bournez and F. Garnier. Proving positive almost-sure termination. In *Proc. RTA '05*, LNCS 3467, pages 323–337, 2005.

[5] M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011.

[6] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012.

[7] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4), 2016. `https://doi.org/10.1145/2866575` Preliminary version appeared in *Proc. TACAS '14*, LNCS 8413, pages 140-155, 2014.

[8] K. Chatterjee, H. Fu, and P. Novotný. Termination analysis of probabilistic programs with martingales. In G. Barthe, J.-P. Katoen, and A. Silva, editors, *Proc. Foundations of Probabilistic Programming*, page 221–258. Cambridge University Press, 2020.

[9] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Trans. Comput. Log.*, 19(3), 2018.

[10] N. Dershowitz. The RTA list of open problems. `https://www.cs.tau.ac.il/~nachum/rtaloop/`.

[11] J. J. Doménech, J. P. Gallagher, and S. Genaim. Control-flow refinement by partial evaluation, and its application to termination and cost analysis. *Theory Pract. Log. Program.*, 19(5-6):990–1005, 2019.

[12] Eclipse. `https://www.eclipse.org/`.

[13] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. Lower bounds for runtime complexity of term rewriting. *J. Autom. Reason.*, 59(1):121–163, 2017. `https://doi.org/10.1007/s10817-016-9397-x` Preliminary version appeared in *Proc. RTA '15*, LIPIcs 36, pages 334-349, 2015.

[14] F. Frohn and J. Giesl. Analyzing runtime complexity via innermost runtime complexity. In *Proc. LPAR '17*, EPiC 46, pages 249–268, 2017. `https://doi.org/10.29007/1nbh`.

[15] F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *Proc. iFM '17*, LNCS 10510, pages 85–101. Springer-Verlag, 2017. `https://doi.org/10.1007/978-3-319-66845-1_6` Winner of the Best Tool Paper Award of the conference.

[16] F. Frohn and J. Giesl. Constant runtime complexity of term rewriting is semi-decidable. *Inf. Process. Lett.*, 139:18–23, 2018. `https://doi.org/10.1016/j.ipl.2018.06.012`.

[3] `https://chc-comp.github.io/`

[17] F. Frohn and J. Giesl. Proving non-termination via loop acceleration. In *Proc. FMCAD '19*, pages 221–230, 2019.

[18] F. Frohn, M. Naaf, M. Brockschmidt, and J. Giesl. Inferring lower runtime bounds for integer programs. *ACM Trans. Program. Lang. Syst.*, 42(3), 2020. https://doi.org/10.1145/3410331 Preliminary version appeared in *Proc. IJCAR '16*, LNCS 9706, pages 550-567, 2016.

[19] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *J. Autom. Reason.*, 37(3):155–203, 2006.

[20] J. Giesl, F. Mesnard, A. Rubio, R. Thiemann, and J. Waldmann. Termination competition (*TermComp* 2015). In *Proc. CADE '15*, LNCS 9195, pages 105–108. Springer-Verlag, 2015. https://doi.org/10.1007/978-3-319-21401-6_6.

[21] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *J. Autom. Reason.*, 58(1):3–31, 2017. https://doi.org/10.1007/s10817-016-9388-y Preliminary version appeared in *Proc. IJCAR '14*, LNAI 8562, pages 184-191, 2014.

[22] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. The termination and complexity competition. In *Proc. TACAS '19*, LNCS 11429, pages 156–166, 2019.

[23] M. Hark, F. Frohn, and J. Giesl. Polynomial loops: Beyond termination. In *Proc. LPAR '20*, EPiC 73, pages 279–297, 2020.

[24] M. W. Haslbeck and R. Thiemann. An Isabelle/HOL formalization of AProVE's termination method for LLVM IR. In *Proc. CPP '21*, pages 238–249, 2021.

[25] J. Hensel, F. Emrich, F. Frohn, T. Ströder, and J. Giesl. AProVE: Proving and disproving termination of memory-manipulating C programs (Competition Contribution). In *Proc. TACAS '17*, LNCS 10206, pages 350–354. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-662-54580-5_21.

[26] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *J. Log. Algebraic Methods Program.*, 97:105–130, 2018. https://doi.org/10.1016/j.jlamp.2018.02.004 Preliminary version appeared in *Proc. SEFM '16*, LNCS 9763, pages 234-252, 2016. Winner of the Silver Medal for the second best paper of the conference.

[27] B. L. Kaminski, J. Katoen, and C. Matheja. Expected runtime analyis by program verification. In G. Barthe, J.-P. Katoen, and A. Silva, editors, *Proc. Foundations of Probabilistic Programming*, page 185–220. Cambridge University Press, 2020.

[28] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*, pages 75–88, 2004.

[29] 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, pages 132–150. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-66167-4_8.

[30] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010.

[31] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS '02*, pages 55–74, 2002.

[32] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and memory safety of C programs (Competition Contribution). In *Proc. TACAS '15*, LNCS 9035, pages 417–419. Springer-Verlag, 2015. https://doi.org/10.1007/978-3-662-46681-0_32.

[33] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *J. Autom. Reason.*, 58(1):33–65, 2017. https://doi.org/10.1007/s10817-016-9389-x Preliminary version appeared in *Proc. IJCAR '14*, LNAI 8562, pages 208-223, 2014.

4 Published Project Results

Publications with scientific quality assurance from 2021 - 2024

[P1] F. Meyer, M. Hark, and J. Giesl. Inferring expected runtimes of probabilistic integer programs using expected sizes. In *Proc. TACAS '21*, LNCS 12651, pages 250-269. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-72016-2_14

[P2] J. Hensel, C. Mensendiek, and J. Giesl. AProVE: Non-termination witnesses for C programs (Competition Contribution). In *Proc. TACAS '22*, LNCS 13244, pages 403-407. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-030-99527-0_21

[P3] 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, pages 193-228. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-08166-8_10

[P4] F. Frohn and C. Fuhs. A calculus for modular loop acceleration and non-termination proofs. *International Journal on Software Tools for Technology Transfer*, 24(5):691-715. Springer-Verlag, 2022. https://doi.org/10.1007/s10009-022-00670-2

[P5] 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, pages 734-754. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-10769-6_43

[P6] F. Frohn and J. Giesl. Proving non-termination and lower runtime bounds with LoAT (System Description). In *Proc. IJCAR '22*, LNCS 13385, pages 712-722. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-031-10769-6_41

[P7] F. Frohn and J. Giesl. ADCL: Acceleration driven clause learning for constrained Horn clauses. In *Proc. SAS '23*, LNCS 14284, pages 259-285. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-44245-2_13

[P8] J. Hensel and J. Giesl. Proving termination of C programs with lists. In *Proc. CADE '23*, LNCS 14132, pages 266-285. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-38499-8_16

[P9] J.-C. Kassing and J. Giesl. Proving almost-sure innermost termination of probabilistic term rewriting using dependency pairs. In *Proc. CADE '23*, LNCS 14132, pages 344-364. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-38499-8_20

[P10] F. Frohn and J. Giesl. Proving non-termination by acceleration driven clause learning (Short Paper). In *Proc. CADE '23*, LNCS 14132, pages 220-233. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-38499-8_13

[P11] N. Lommen and J. Giesl. Targeting completeness: Using closed forms for size bounds of integer programs. In *Proc. FroCoS '23*, LNCS 14279, pages 3-22. Springer-Verlag, 2023. https://doi.org/10.1007/978-3-031-43369-6_1

[P12] M. Hark, F. Frohn, and J. Giesl. Termination of triangular polynomial loops. *Formal Methods in System Design*, Springer-Verlag. 2023. https://doi.org/10.1007/s10703-023-00440-z Preliminary version appeared in *Proc. SAS '20*, LNCS 12389, pages 89-112. Springer-Verlag, 2021. https://doi.org/10.1007/978-3-030-65474-0_5

[P13] J.-C. Kassing, F. Frohn, and J. Giesl. From innermost to full almost-sure termination of probabilistic term rewriting. In *Proc. FoSSaCS '24*, LNCS 14575, pages 206-228. Springer, 2024. https://doi.org/10.1007/978-3-031-57231-9_10

[P14] J.-C. Kassing, S. Dollase, and J. Giesl. A complete dependency pair framework for almost-sure innermost termination of probabilistic term rewriting. In *Proc. FLOPS '24*, LNCS 14659, pages 62-80. Springer-Verlag, 2024. https://doi.org/10.1007/978-981-97-2300-3_4

[P15] N. Lommen, E. Meyer, and J. Giesl. Control-flow refinement for complexity analysis of probabilistic programs in KoAT (Short Paper). In *Proc. IJCAR '24*, LNCS. Springer-Verlag, 2024. To appear. Extended version appeared at *arXiv*, CoRR abs/2402.03891. https://doi.org/10.48550/arXiv.2402.03891

[P16] J.-C. Kassing, G. Vartanyan, and J. Giesl. A dependency pair framework for relative termination of term rewriting. In *Proc. IJCAR '24*, LNCS. Springer-Verlag, 2024. To appear. Extended version appeared at *arXiv*, CoRR abs/2404.15248. https://doi.org/10.48550/arXiv.2404.15248

[P17] F. Frohn and J. Giesl. Satisfiability modulo exponential integer arithmetic. In *Proc. IJCAR '24*, LNCS. Springer, 2024. To appear. Extended version at *arXiv*, CoRR abs/2402.01501. https://doi.org/10.48550/arXiv.2402.01501

[P18] F. Baader and J. Giesl. On the complexity of the small term reachability problem for terminating term rewriting systems. In *Proc. FSCD '24*, LIPIcs, 2024. To appear. https://verify.rwth-aachen.de/giesl/papers/FSCD2024.pdf

[P19] F. Frohn and J. Giesl. Integrating loop acceleration into bounded model checking. In *Proc. FM '24*, LNCS. Springer-Verlag, 2024. To appear. Extended version appeared at *arXiv*, CoRR abs/2401.09973. https://doi.org/10.48550/arXiv.2401.09973

[P20] F. Emrich, J. Hensel, and J. Giesl. AProVE: Modular termination analysis of memory-manipulating C programs. In D. Beyer (ed.), *Automatic Software Verification*, Springer-Verlag, 2024. To appear. Extended version appeared at *arXiv*, CoRR 2302.02382. https://doi.org/10.48550/arXiv.2302.02382

Publications from the first funding period: [7, 13–16, 18, 20, 21, 25, 26, 29, 32, 33]

In addition to the publications, all our technical reports are available on *arXiv* (see https://arxiv.org/archive/cs).