





AProVE (KoAT + LoAT)

(Competition Contribution)

Nils Lommen(✉) and Jürgen Giesl(✉)

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

Abstract. To (dis)prove termination of C programs, AProVE uses symbolic execution to transform the program's LLVM code into an integer transition system (ITS). These ITSs are analyzed by our backend tools KoAT (for termination) and LoAT (for non-termination) which we integrated into our novel framework to replace previously used external backend tools. In this way, we benefit from the recent improvements in the backend tools KoAT and LoAT. The transformation steps in AProVE and the tools in the backend produce sub-proofs which are then combined automatically in order to generate a complete termination proof. If non-termination is proved, then a witness for a non-terminating path in the original C program is returned.

1 Verification Approach and Software Architecture

AProVE (KoAT + LoAT) is a framework combining our three tools AProVE, KoAT, and LoAT to prove or disprove termination of C programs automatically. To this end, the C program is compiled into the intermediate representation of the LLVM framework [28] by the Clang compiler [1]. Afterwards, the LLVM program is processed by AProVE and transformed into a *symbolic execution graph* (SEG, see [24, 26, 36, 37] for more details). Finally, the SEG is either analyzed directly by AProVE (see [23–25, 37] for more details on the internal (non-)termination proofs in AProVE) or transformed into *integer transition systems* (ITSs). These ITSs are analyzed by our tools KoAT [8, 20, 29–32] (for termination) and LoAT [14–17] (for non-termination). In case of non-termination, the proofs by AProVE and LoAT are transformed into non-termination proofs for the original C program.

Earlier versions of AProVE that participated in *SV-COMP* until 2022 used the external tool T2 [7] for proving termination of ITSs (and both T2 and LoAT for proving non-termination). In contrast, instead of T2, our new version uses our own ITS analyzer KoAT in the backend which allows us to benefit from the numerous recent improvements that we developed for KoAT. A bird's-eye view of our approach is sketched in Fig. 1.

Termination Analysis by KoAT: In the following, we briefly describe how our tool KoAT proves termination of ITSs which result from the transformation of C programs. LoAT can be used to *disprove* termination and infer *lower* run-

N. Lommen—Jury member representing AProVE (KoAT + LoAT) at *SV-COMP 2025*

© The Author(s) 2025
A. Gurfinkel and M. Heule (Eds.): TACAS 2025, LNCS 15698, pp. 205–211, 2025.
https://doi.org/10.1007/978-3-031-90660-2_13

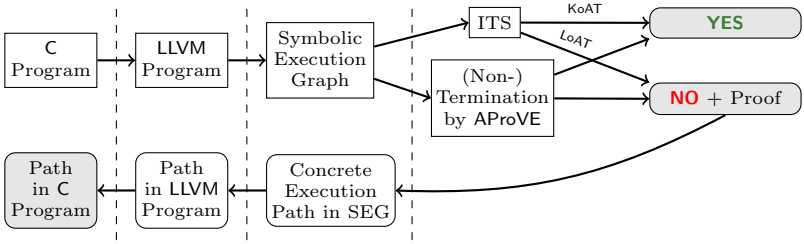


Fig. 1. AProVE (KoAT + LoAT) Framework for Proving and Disproving Termination

time bounds for ITSs, and its integration in order to prove non-termination of C programs was described in [25]. In contrast, KoAT is a tool to automatically *prove* termination and infer *upper* complexity bounds for ITSs based on a modular analysis of separate program parts [8, 32]. To prove termination of sub-programs, it uses *multiphase-linear ranking functions* (MΦRFs) [5, 20]. In contrast to classical ranking functions, MΦRFs can also represent bounds on programs with multiple "phases" of executions.

Moreover, we embedded a technique [21] to analyze termination of so-called *triangular weakly non-linear loops* (twn-loops) in our tool KoAT [29, 32]. In particular, this approach also allows us to analyze programs with *non-linear* arithmetic. An example for a terminating twn-loop, which may result from a sub-program within a larger C program, is:

$$\textbf{while } (x_2 - x_1^2 > 0 \wedge x_1 > 0) \textbf{ do } (x_1, x_2, x_3) \leftarrow (4 \cdot x_1,\ 9 \cdot x_2 - 8 \cdot x_3^3,\ x_3) \qquad (1)$$

This loop does not admit a MΦRF over $\mathbb{R}$ (see [22]). 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). For example, a loop whose body consists of the assignment $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is triangular, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_1 + 1)$ is not triangular. From a practical point of view, the restriction to triangular loops seems quite natural. For example, in [15], 1511 polynomial loops were extracted from the *Termination Problems Data Base* [38], the benchmark collection which is used at the annual *Termination and Complexity Competition* [19], 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 occurrence in its own update. So for example, a loop with the body $(x_1, x_2) \leftarrow (x_1 + x_2^2, x_2 + 1)$ is weakly non-linear, whereas a loop with the body $(x_1, x_2) \leftarrow (x_1 \cdot x_2, x_2 + 1)$ is not. With triangularity and weak non-linearity, one can compute a *closed form* which corresponds to applying the loop's update n times. For example, the closed forms of (1) are $x_1 \cdot 4^n$, $(x_2 - x_3^3) \cdot 9^n + x_3^3$, and x_3 for x_1, x_2, and x_3, respectively.

Using these closed forms, termination can be reduced to a formula over $\mathbb{Z}$ [21] (whose satisfiability is decidable for linear arithmetic and where SMT solvers often also prove (un)satisfiability in the non-linear case). The idea of the reduction is to consider each atom of the loop guard individually, instantiate the variables in the atoms by their closed forms, and order the summands of the resulting expressions w.r.t. their growth rate in n. For example, for the atom $x_2 - x_1^2 > 0$ of (1), we obtain $\alpha_1 \cdot 16^n + \alpha_2 \cdot 9^n + \alpha_3 > 0$ where $\alpha_1 = -x_1^2$, $\alpha_2 = x_2 - x_3^3$, and $\alpha_3 = x_3^3$. If the initial values of the variables satisfy $\alpha_1 > 0$, then the addend $\alpha_1 \cdot 16^n$ would dominate all other addends at some point and the atom $x_2 - x_1^2 > 0$ would hold for all large enough n, assuming that there are no overflows. (However, in our example $\alpha_1 = -x_1^2 > 0$ is unsatisfiable.) Otherwise, if $\alpha_1 = -x_1^2 = 0$, then the second addend $\alpha_2 \cdot 9^n$ is the fastest growing summand. Applying this idea to all addends subsequently, i.e., checking if the first polynomials $\alpha_1, \ldots, \alpha_{j-1}$ are 0 and α_j is positive, we can reduce non-termination to an existential first-order (FO) problem. For our example, we obtain the reduction $\mathrm{red}(x_2 - x_1^2 > 0) = \alpha_1 > 0 \vee (\alpha_1 = 0 \wedge \alpha_2 > 0) \vee (\alpha_1 = 0 \wedge \alpha_2 = 0 \wedge \alpha_3 > 0)$ for the atom $x_2 - x_1^2 > 0$. Similarly, we have $\mathrm{red}(x_1 > 0) = (x_1 > 0)$ for the second atom of the loop guard. Replacing the atoms of the guard by their reduction results in the overall FO problem $\psi = \mathrm{red}(x_2 - x_1^2 > 0) \wedge \mathrm{red}(x_1 > 0)$. Thus, (1) is non-terminating over $\mathbb{Z}$ iff ψ is satisfiable over $\mathbb{Z}$. In our example, the formula ψ is unsatisfiable since $\mathrm{red}(x_2 - x_1^2 > 0)$ is only satisfiable if $x_1 = 0$. However, this violates $\mathrm{red}(x_1 > 0)$. Thus, the twn-loop (1) is terminating.

KoAT's novel approach of analyzing some sub-programs with ranking functions and other sub-programs with our technique for twn-loops increases the power of automated termination analysis substantially, in particular also for programs containing non-linear arithmetic. Nevertheless, there still exist programs whose termination is difficult to analyze and where it would help to gain "more information" on the values of variables in order to determine the infeasibility of certain paths in the program. For example, one could transform a loop whose body contains an if-else-instruction where either always the if-branch or the else-branch are executed into two separate single-path loops. This transformation explicitly removes infeasible paths which contain both branches. Thus, the idea is to transform an ITS $\mathcal{P}$ into a new ITS $\mathcal{P}'$ which is "easier" to analyze. Of course, we ensure that the runtime of $\mathcal{P}'$ is *at least* the runtime of $\mathcal{P}$. Then it is sound to prove termination for $\mathcal{P}'$ instead of $\mathcal{P}$. Such transformations can be performed by *control-flow refinement* via partial evaluation [11], which we integrated into KoAT's approach [20, 31].

2 Discussion of Strengths and Weaknesses

An underlying design concept of AProVE (KoAT + LoAT) is its modular structure. It allows us to use different backends and techniques to analyze programs and benefit from their individual strengths. Hence, our new framework can make use of recent progress in software verification tools. Furthermore, both AProVE

and our backend tools KoAT and LoAT are modular themselves (see, e.g., KoAT's modular analysis of sub-programs in Sect. 1).

One of AProVE's main weaknesses was that it essentially relied on variants of (linear) ranking functions for termination proofs. However, this problem has now been solved by integrating our tool KoAT into AProVE. Besides ranking functions, KoAT also uses a technique for termination analysis of twn-loops which allows us to analyze programs with non-linear arithmetic on which the other sound tools participating in the termination category of *SV-COMP* fail. For example, the C program which just instantiates all variables non-deterministically and consists of the loop (1) and even its following linear, simplified variant from [22]

$$\textbf{while } (x_1 < x_2 \wedge x_1 > 0) \textbf{ do } (x_1, x_2) \leftarrow (3 \cdot x_1, 2 \cdot x_2)$$

can both not be shown terminating by Ultimate Automizer [10] and 2LS [35], which were the two most powerful sound[1] tools at the termination category of last year's *SV-COMP*. In contrast, AProVE proves their termination using the implementation of our termination technique for twn-loops in KoAT.

Concerning the proofs of non-termination, currently AProVE (KoAT + LoAT) uses a version of LoAT which disproves termination using the loop acceleration technique of [16]. In [17], a new version of LoAT was developed that is based on an acceleration driven clause learning calculus which improves LoAT's power for non-termination proofs substantially. We plan to integrate this new version in our AProVE (KoAT + LoAT) framework in the future, using a new format to ease the automated handling of LoAT's non-termination proofs. Moreover, as KoAT and LoAT can also analyze complexity of programs, it would be interesting to extend AProVE (KoAT + LoAT) to complexity analysis as well.

3 Setup and Configuration

AProVE (KoAT + LoAT) is developed in the "*Programming Languages and Verification*" group at RWTH Aachen University. A list of present and past contributors can be accessed on the website [2]. In *SV-COMP 2025*, AProVE (KoAT + LoAT) only participates in the category "*Termination*". All files from the submitted archive [33] must be extracted into one folder. AProVE is implemented in Java and needs a Java 17 Runtime Environment. To analyze the resulting ITSs in the backend, the tools KoAT [20, 29–32] and LoAT [14–17] are used. Furthermore, it applies the satisfiability checkers Z3 [9], Yices [12], and MiniSAT [13], see [18]. Our archive contains all these tools. Using the wrapper script `aprove.py` in the BenchExec repository, AProVE (KoAT + LoAT) can be invoked, e.g., on the benchmarks defined in `aprove.xml` in the *SV-COMP* repository. The most recent version of our tool for *SV-COMP 2025* [6] can be downloaded at [33]. Moreover, C programs for all examples from the paper are available at [33] and [34].

[1] The tool PROTON [27] (which won the termination category last year) outputs termination for both these loops, but also for the *non-terminating* variants of these loops where the guard $x_1 > 0$ is missing. In contrast, both Ultimate Automizer and 2LS prove non-termination of these variants.

Data Availability Statement. Our tools are available at their respective websites (for AProVE [2], KoAT [3], and LoAT [4]). The version of AProVE (KoAT + LoAT) used for *SV-COMP 2025* is archived at Zenodo [33].

References

[1] Clang: https://clang.llvm.org.

[2] AProVE Website: https://aprove.informatik.rwth-aachen.de/.

[3] KoAT Website: https://koat.verify.rwth-aachen.de/.

[4] LoAT Website: https://loat-developers.github.io/LoAT/.

[5] 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.

[6] D. Beyer and J. Strejček. "Improvements in Software Verification and Witness Validation: *SV-COMP 2025*". In: *Proc. TACAS '25*. LNCS. 2025.

[7] 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.

[8] 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.

[9] 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.

[10] D. Dietsch, M. Bentele, M. Ebbinghaus, M. Heizmann, D. Klumpp, A. Podelski, and F. Schüssele. Ultimate Automizer *SV-COMP 2025*. Zenodo. 2024. DOI: 10.5281/zenodo.14209043.

[11] 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.

[12] B. Dutertre and L. de Moura. *System Description:* Yices 1.0. https://yices.csl.sri.com/papers/yices-smtcomp06.pdf. 2006.

[13] N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Proc. SAT '03*. LNCS 2919. 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

[14] F. Frohn and J. Giesl. "Proving Non-Termination via Loop Acceleration". In: *Proc. FMCAD '19*. 2019, pp. 221–230. DOI: 10.23919/FMCAD.2019.8894271.

[15] 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.

[16] 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.

[17] F. Frohn and J. Giesl. “Proving Non-Termination by Acceleration Driven Clause Learning (Short Paper)”. In: *Proc. CADE ’23*. LNCS 14132. 2023, pp. 220–233. DOI: 10.1007/978-3-031-38499-8_13.

[18] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. “SAT Solving for Termination Analysis with Polynomial Interpretations”. In: *Proc. SAT ’07*. LNCS 4501. 2007, pp. 340–354. DOI: 10.1007/978-3-540-72788-0_33.

[19] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. “The Termination and Complexity Competition”. In: *Proc. TACAS ’19*. LNCS 11429. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[20] 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.

[21] M. Hark, F. Frohn, and J. Giesl. “Termination of Triangular Polynomial Loops”. In: *Formal Methods in System Design* (2023). DOI: 10.1007/s10703-023-00440-z.

[22] 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.

[23] 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. 2017, pp. 350–354. DOI: 10.1007/978-3-662-54580-5_21.

[24] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. “Termination and Complexity Analysis for Programs with Bitvector Arithmetic by Symbolic Execution”. In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 105–130. URL: https://doi.org/10.1016/j.jlamp.2018.02.004.

[25] J. Hensel, C. Mensendiek, and J. Giesl. “AProVE: Non-Termination Witnesses for C Programs (Competition Contribution)”. In: *Proc. TACAS ’22*. LNCS 14132. 2022, pp. 403–407. DOI: 10.1007/978-3-030-99527-0_21.

[26] J. Hensel and J. Giesl. “Proving Termination of C Programs with Lists”. In: *Proc. CADE ’23*. LNCS 14132. 2023, pp. 266–285. DOI: 10.1007/978-3-031-38499-8_16.

[27] H. Karmarkar, M. Kumar, R. Metta, and D. Mukhopadhyay. PROTON *SV-COMP 2025*. Zenodo. 2024. DOI: 10.5281/zenodo.14209458.

[28] C. Lattner and V. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proc. CGO ’04*. 2004, pp. 75–88. URL: https://doi.org/10.1109/CGO.2004.1281665.

[29] 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.

[30] 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.

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

[32] N. Lommen, É. Meyer, and J. Giesl. "Targeting Completeness: Automated Complexity Analysis of Integer Programs". In: *CoRR* abs/2412.01832 (2024). DOI: 10.48550/arXiv.2412.01832.

[33] N. Lommen and J. Giesl. AProVE (KoAT + LoAT) Download. Zenodo. 2024. DOI: 10.5281/zenodo.13937818.

[34] N. Lommen and J. Giesl. AProVE (KoAT + LoAT) Website. 2025. URL: https://koat.verify.rwth-aachen.de/svcomp25.

[35] V. Malík, F. Nečas, P. Schrammel, and T. Vojnar. 2LS. Zenodo. 2023. DOI: 10.5281/zenodo.10184626.

[36] 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. 2015, pp. 417–419. DOI: 10.1007/978-3-662-46681-0_32.

[37] 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". In: *Journal of Automated Reasoning* 58.1 (2017), pp. 33–65. DOI: 10.1007/s10817-016-9389-x.

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

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

