AProVE: Non-Termination Witnesses for C Programs*
(Competition Contribution)

Jera Hensel✉, Constantin Mensendiek, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. To (dis)prove termination of C programs, AProVE uses symbolic execution to transform the program's LLVM code into an integer transition system, which is then analyzed by several backends. The transformation steps in AProVE and the tools in the backend only produce sub-proofs in their domains. Hence, we now developed new techniques to automatically combine the essence of these proofs. If non-termination is proved, then they yield an overall witness, which identifies a non-terminating path in the original C program.

1 Verification Approach and Software Architecture

To prove (non-)termination of a C program, AProVE uses the Clang compiler [7] to translate it to the intermediate representation of the LLVM framework [15]. Then AProVE symbolically executes the LLVM program and uses abstraction to obtain a finite symbolic execution graph (SEG) containing all possible program runs. We refer to [14,17] for further details on our approach to prove termination.

To prove non-termination, AProVE runs three approaches in parallel, see Fig. 1. The first two approaches transform the lassos of the SEG to integer transition systems (ITSs), which are then passed to the tools T2 [6] and LoAT [11]. If one of the tools returns a proof of non-termination, AProVE uses it to construct a non-terminating path through the C program. The path of the first succeeding approach is returned to the user while all other computations are stopped. T2's proof consists of a recurrent set characterizing those variable assignments that lead to a non-terminating ITS run. Here, AProVE uses an SMT solver to identify a corresponding concrete assignment of the variables in the ITS (which correspond to the variables in the (abstract) program states of the SEG). The third approach transforms the lassos of the SEG directly to SMT formulas which are only satisfiable if there is a non-terminating path, and in this case, we can deduce a variable assignment from the model of the formulas returned by the solver. While the first and the third approach were already available in AProVE before [13], we now extended them by the generation of non-termination witnesses. To this end, the variable assignment obtained from these approaches

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

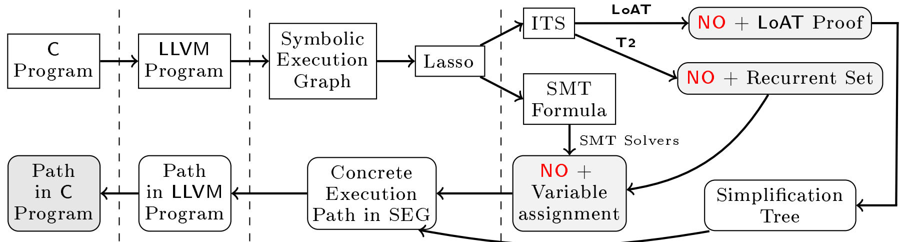

Fig. 1: AProVE's Workflow for Non-Termination Analysis

is used by AProVE to step through the corresponding lasso of the SEG in order to obtain a concrete execution path which witnesses non-termination. To ensure that the generation of the path terminates, AProVE stops as soon as a program state of the SEG is visited twice. Thus, this approach only succeeds if the first loop on the path whose body is executed several times is already the non-terminating loop. However, it does not find non-termination witnesses for programs with several loops, where the non-terminating path first leads through several iterations of other loops before it ends in a non-terminating loop.

To handle such programs as well, we now developed a novel second approach for proving non-termination which uses our tool LoAT in the backend. To understand how LoAT finds non-termination proofs, consider the function `f` in Fig. 2. The first loop decrements `x` as long as `x` is positive and increments `y` by the same amount. Afterwards, the second loop does not terminate if `y` is greater than 1. Hence, the function `f` does not terminate if the initial value of the parameter `x` is greater than 1. LoAT can detect such coherences in the corresponding ITS (Fig. 3a) generated by AProVE. To this end, LoAT uses different forms of *loop acceleration*: *Finite acceleration* combines several iterations of a looping rule into a new rule. LoAT applies this simplification to the rule r_1 representing the first loop, resulting in the new rule r_4 in Fig. 3b. In the second looping rule r_3, the guard is invariant w.r.t. the update of the variables in this rule. In such a case, LoAT applies *non-terminating acceleration*, transforming r_3 to r_5. Finally, *chaining* allows to represent the successive execution of two rules. For example, the rule r_6 is the result of chaining r_0 and r_4. The exact simplification steps performed by LoAT in this example are shown in Fig. 3c. Note that the final rule r_8 starts from the initial function

```
void f(x,y) {
  y = 0;
  while (x > 0) {
    x = x-1;
    y = y+1;
  }
  while (y > 1)
    y = y;
}
```

Fig. 2: Example C Function

$$
\begin{array}{lll}
r_0\colon f(x,y) & \to \ell_1(x,0) & \\
r_1\colon \ell_1(x,y) & \to \ell_1(x-1,y+1) & [x>0] \\
r_2\colon \ell_1(x,y) & \to \ell_2(x,y) & [x\leq 0] \\
r_3\colon \ell_2(x,y) & \to \ell_2(x,y) & [y>1]
\end{array}
$$

Fig. 3a: Corresponding ITS

$$
\begin{array}{lll}
r_4\colon \ell_1(x,y) & \to \ell_1(0,y+x) & [x>0] \\
r_5\colon \ell_2(x,y) & \to \infty & [y>1] \\
r_6\colon f(x,y) & \to \ell_1(0,x) & [x>0] \\
r_7\colon f(x,y) & \to \ell_2(0,x) & [x>0] \\
r_8\colon f(x,y) & \to \infty & [x>1]
\end{array}
$$

Fig. 3b: Simplified Rules

symbol and directly goes to non-termination. Every variable assignment satisfying the respective final guard $x > 1$ results in a non-terminating run.

The simplification tree in Fig. 3c is also the starting point for our new technique to generate non-termination witnesses. AProVE constructs this tree from LoAT's proof output. Then, by processing the leaves of the simplification tree from left to right, a path through the SEG can be derived. To determine how often one has to traverse earlier loops on the path to the non-terminating loop, AProVE uses an SMT solver to find a concrete variable assignment that satisfies the final guard. In our example, the final guard $x > 1$ would be satisfied by $\{x = 2, y = 0\}$, for example. Consequently, the corresponding concrete execution path includes two iterations of the first loop before reaching the non-terminating second loop.

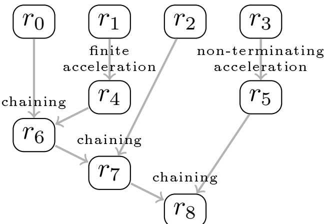

Fig. 3c: Simplification Tree

Once the path is constructed, AProVE extracts the LLVM program positions from the states, obtaining a non-terminating path through the LLVM program in form of a lasso. Using the Clang debug information output, AProVE then matches the LLVM lines to the lines in the C program. The resulting C witness can be validated by the tools CPAchecker [5] and Ultimate Automizer [12].

2 Discussion of Strengths and Weaknesses

In general, AProVE is especially powerful on programs where a precise modeling of the values of program variables and memory contents is needed to (dis)prove termination. However, on large programs containing many variables which are not relevant for termination, tools with CEGAR-based approaches are often faster. The reason is that AProVE does not implement any techniques to decide which variables are relevant for (non-)termination.

Furthermore, one of AProVE's most crucial weaknesses when proving non-termination in past editions of *SV-COMP* was to produce a meaningful witness. Therefore, in the two approaches for proving non-termination in AProVE that are based on T2 or on the direct analysis of lassos of the SEG, we added the novel techniques presented in the current paper to generate non-termination witnesses from the obtained variable assignments. Here, the problem is that when computing a concrete execution path, we cannot be sure when to stop the computation: Whenever we visit a program position repeatedly, we do not know if this position is part of the non-terminating loop of the lasso, or if it is still part of the finite path to the non-terminating loop.

In contrast, in our new approach based on LoAT, the simplification tree allows us to infer the order in which the loops of the program are traversed and this tree also contains the information which loop is the non-terminating one. Thus, this approach extends AProVE's power substantially, since it can find non-termination witnesses for programs where all non-terminating paths lead through several iterations of more than one loop. On the other hand, there are

also examples where the other two approaches outperform the approach based on LoAT, e.g., if T2 finds a non-termination proof and LoAT does not. Our observation is that especially for small programs containing only a single loop, the other approaches are often faster. This is also confirmed by our results in the *Termination* category of *SV-COMP 2022*: While in the sub-categories *MainControlFlow* and *MainHeap*, 83% of the non-termination proofs are found using T2 or the direct SMT approach, in *Termination-Other*, 95% of the non-termination proofs result from the LoAT approach. This set consists of especially large programs, which often contain more than one loop.

More information about *SV-COMP 2022* including the competition results can be found in the competition report [3].

3 Setup and Configuration

AProVE is developed in the "*Programming Languages and Verification*" group headed by J. Giesl at RWTH Aachen University. On the web site [2], AProVE can be downloaded or accessed via a web interface. Moreover, [2] also contains a list of external tools used by AProVE and a list of present and past contributors.

In *SV-COMP 2022*, AProVE only participates in the category "*Termination*". All files from the submitted archive must be extracted into one folder. AProVE is implemented in Java and needs a Java 11 Runtime Environment. Moreover, AProVE requires the Clang compiler [7] to translate C to LLVM. To analyze the resulting ITSs in the backend, AProVE uses LoAT [11] and T2 [6]. Furthermore, it applies the satisfiability checkers Z3 [8], Yices [9], and MiniSAT [10] in parallel (our archive contains all these tools). As a dependency of T2, Mono [16] (version ≥ 4.0) needs to be installed. Extending the path environment is necessary so that AProVE can find these programs. Using the wrapper script `aprove.py` in the `BenchExec` repository, AProVE can be invoked, e.g., on the benchmarks defined in `aprove.xml` in the *SV-COMP* repository. The most recent version of AProVE with the improved witness generation can be downloaded at [1].

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [3] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [4].

References

1. AProVE: https://github.com/aprove-developers/aprove-releases/releases
2. AProVE Website: https://aprove.informatik.rwth-aachen.de/
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS '22. LNCS (2022)
4. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022), https://doi.org/10.5281/zenodo.5959149

5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV '11. pp. 184–190. LNCS 6806 (2011), https://doi.org/10.1007/978-3-642-22110-1_16
6. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: Temporal property verification. In: Proc. TACAS '16. pp. 387–393. LNCS 9636 (2016), https://doi.org/10.1007/978-3-662-49674-9_22
7. Clang: https://clang.llvm.org
8. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS '08. pp. 337–340. LNCS 4963 (2008), https://doi.org/10.1007/978-3-540-78800-3_24
9. Dutertre, B., de Moura, L.: System Description: Yices 1.0 (2006), https://yices.csl.sri.com/papers/yices-smtcomp06.pdf
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT '03. pp. 502–518. LNCS 2919 (2003), https://doi.org/10.1007/978-3-540-24605-3_37
11. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Proc. FMCAD '19. pp. 221–230 (2019), https://doi.org/10.23919/FMCAD.2019.8894271
12. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate Automizer with array interpolation. In: Proc. TACAS '15. pp. 455–457. LNCS 9035 (2015), https://doi.org/10.1007/978-3-662-46681-0_43
13. Hensel, J., Emrich, F., Frohn, F., Ströder, T., Giesl, J.: AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution). In: Proc. TACAS '17. pp. 350–354. LNCS 10206 (2017), https://doi.org/10.1007/978-3-662-54580-5_21
14. Hensel, J., Giesl, J., Frohn, F., Ströder, T.: Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. Journal of Logical and Algebraic Methods in Programming **97**, 105–130 (2018), https://doi.org/10.1016/j.jlamp.2018.02.004
15. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO '04. pp. 75–88 (2004), https://doi.org/10.1109/CGO.2004.1281665
16. Mono: https://www.mono-project.com/
17. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. of Aut. Reasoning **58**(1), 33–65 (2017), https://doi.org/10.1007/s10817-016-9389-x

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (https://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.