# AProVE: Proving and Disproving Termination of Memory-Manipulating C Programs*
## (Competition Contribution)

J. Hensel**, F. Emrich, F. Frohn, T. Ströder, and J. Giesl

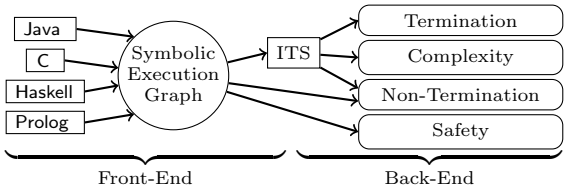
LuFG Informatik 2, RWTH Aachen University, Germany

**Abstract.** AProVE is a system for automatic termination and complexity analysis of C, Java, Haskell, Prolog, and several forms of rewrite systems. The new contributions in this version of AProVE are its capabilities to prove non-termination of C programs and to handle recursive C programs, even if these programs use pointer arithmetic combined with direct memory accesses. Moreover, in addition to mathematical integers, AProVE can now also handle fixed-width bitvector integers.

## 1 Verification Approach and Software Architecture

The focus of AProVE's analysis for C programs lies on the connection between memory addresses and their contents. To this end, AProVE employs symbolic execution and abstraction to obtain a finite *symbolic execution graph* from a C program. This graph over-approximates all possible program executions and models memory addresses and contents explicitly. However, all reasoning required to construct this graph is reduced to first-order SMT solving on integers. During the construction of the graph, AProVE proves that the original program does not expose undefined behavior. For proving termination, the strongly connected components (SCCs) of the graph are transformed to integer transition systems (ITSs). Standard techniques can be used to analyze termination of these ITSs and in case of success, this implies termination of the original program. For more information on AProVE's approach to prove termination of C programs, we refer to [15]. Moreover, AProVE's modular architecture allows to use the same backend to prove termination for several programming languages (cf. the figure below). An overview on the use of AProVE for different languages is found in [10].

The approach of [15] is powerful for *termination* of C, but we need several adaptions for *non-termination*, as both the symbolic execution graph and the resulting ITSs are over-approximations. So in general, non-termination of an ITS does not imply non-termination of the original program. However, there are many



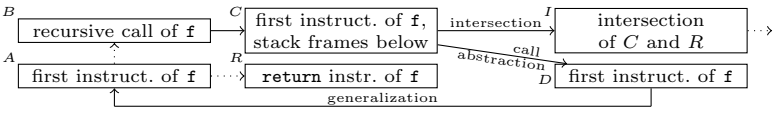

* Supported by DFG grant GI 274/6-1.

** Jury member. E-Mail: hensel@informatik.rwth-aachen.de

program instructions that are modeled precisely in the graph and in the resulting ITSs. Therefore, to prove non-termination of the program, it suffices to find a non-terminating lasso of the graph that does not contain any proper over-approximation. Here, a lasso is an SCC together with a path from the root of the graph to the SCC. AProVE's back-end does not consider that evaluation of ITSs may only begin with designated "start terms" (in order to exclude spurious symbolic execution paths). Thus, to prove non-termination of the resulting ITS, we use the tool T2 [4] which takes such start terms into account. Moreover, we heuristically add conditions to the ITS rules which restrict the possible values of the variables (i.e., they yield an under-approximation of the ITS). Then, non-termination of the under-approximated ITS implies non-termination of the program.

In addition, we implemented an alternative approach for non-termination which uses our over-approximation of the program to detect *candidates* for non-terminating executions. Afterwards, one still has to prove that the candidate corresponds to an actual execution of the program. To this end, we build SMT formulas for the cycles in the symbolic execution graph. They encode that those program variables and memory contents which influence the control flow are not changed when traversing the cycle. A model $M_1$ of such a formula $\varphi_1$ corresponds to actual values where a loop in the program is not left. Then, this model needs to be traced back to the initial state of the graph. For this, the path from the initial state to the cycle is transformed into an SMT formula $\varphi_2$, where the values in the cycle are chosen according to the model $M_1$. A model of $\varphi_2$ yields concrete input values for the initial state that lead to a non-terminating execution. This approach is based on a previous technique in AProVE for proving non-termination of Java programs [3]. Since both our approaches to prove non-termination are orthogonal in power, these approaches are run in parallel in AProVE.

We also extended our graph construction of [15] to support recursive programs. To this end, we adapted our techniques developed for recursive Java programs [2] to handle explicit (de)allocation of memory and pointer arithmetic. (Compared to [2], a particular challenge is to infer and exploit information about memory that is not reachable from program variables.) The nodes of the symbolic execution graph are *abstract states*, which represent sets of concrete program states. To prove termination of a function `f`, we start with a state $A$ whose program position is at `f`'s initial instruction. If $A$ evaluates to a state $B$ where `f` is called recursively, this yields a next state $C$ where a new stack frame at `f`'s initial instruction is added on top of the stack of $B$ (we refer to $C$ as a "*call state*"). To ensure termination of the graph construction, we perform *call abstraction*, which leads to a state $D$ that results from $C$ by removing all lower stack frames except the top one. Our previous state $A$ is a *generalization* of $D$, i.e., all concrete states represented by $D$ are also represented by $A$. Thus, we do not need further symbolic execution for the less general state $D$. However, whenever the initial state $A$ evaluates to a *return state* $R$ where the function `f` terminates, we have to take into account that the



call of `f` in state $C$ might lead to such a return state. Thus, for every pair of a call state $C$ and a return state $R$ of `f`, we construct an *intersection* state $I$ which represents those states that result from $C$ after completely executing the call of `f` in its topmost stack frame. With this extension, the symbolic execution graph construction of [15] can now also deal with recursion.

Finally, while up to now we assumed the program variables to range over mathematical integers $\mathbb{Z}$, we now developed an extension which also allows to handle fixed-width bitvector integers, cf. [11]. So our technique for termination analysis of C programs now covers both byte-accurate pointer arithmetic and bit-precise modeling of integers. To this end, we express relations between bitvectors by corresponding relations on $\mathbb{Z}$. In this way, we can use standard SMT solving over $\mathbb{Z}$ for all steps needed to construct the symbolic execution graph. Moreover, this allows us to obtain ITSs over $\mathbb{Z}$ from these graphs, and to use standard approaches for termination analysis of these ITSs.

## 2 Strengths and Weaknesses

Our approach is particularly powerful when the control flow depends on relations between addresses and memory contents. In addition, AProVE also proves absence of undefined behavior while many other termination analyzers just *assume* memory safety when analyzing C programs. AProVE's participation at former editions of *SV-COMP* and at the annual *Termination Competition*[1] shows the applicability of our approach to termination analysis of real-world programming languages: AProVE won most categories related to termination of C, Java, Haskell, Prolog, and to termination or runtime complexity of rewriting.

The downside of our approach is that it often takes long to construct symbolic execution graphs and that AProVE cannot give any meaningful answer before this construction is finished. Thus, AProVE's runtime is often higher than that of other tools. Moreover, our approach is currently limited to programs operating on integers and pointers (including arrays) but without `struct` types. For `struct` types, a main challenge for future work is to extend our approach to handle recursive data types in combination with explicit low-level pointer arithmetic.

## 3 Setup and Configuration

Since the setup of AProVE has not changed much during the last years, this section is mainly a recapitulation of the corresponding section in [14]. AProVE is developed in the "*Programming Languages and Verification*" group headed by Jürgen Giesl at RWTH Aachen University. On the website [1], AProVE can be obtained as a command-line tool or as a plug-in for the popular Eclipse software development environment [8]. In this way, AProVE can already be applied during program construction. Moreover, AProVE can be accessed directly via a web interface as well. The website [1] also contains a list of external tools used by AProVE and a list of present and past contributors.

[1] `http://www.termination-portal.org/wiki/Termination_Competition`

The particular version for analyzing C programs according to the *SV-COMP* format can be downloaded from the following URL. AProVE only participates in the category "*Termination*". Thus, in this version of AProVE, we disabled some checks for memory safety, since it was agreed that only memory safe programs will be included in the termination category of *SV-COMP*.

`http://aprove.informatik.rwth-aachen.de/eval/Pointer/AProVE2017.zip`

All files from this archive must be extracted into one folder. AProVE is implemented in Java and needs a Java 8 Runtime Environment. To avoid handling the intricacies of C, we analyze programs in the intermediate representation of the LLVM compilation framework [12] and AProVE requires the Clang compiler [5] (version $\geq 3.5$) to translate C to LLVM. To solve the search problems in the back-end, AProVE uses T2 and it applies the satisfiability checkers Z3 [6], Yices [7], and MiniSAT [9] in parallel (our archive contains all these tools). As a dependency of T2, Mono [13] (version $\geq 4.0$) needs to be installed. Extending the path environment is necessary so that AProVE can find these programs. AProVE can be invoked using the wrapper script `aprove.py` in the BenchExec tool.

## References


1. AProVE. `http://aprove.informatik.rwth-aachen.de/`.
2. M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. *RTA '11*, pp. 155–170.
3. M. Brockschmidt, T. Ströder, C. Otto, J. Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. *FoVeOOS '11*, pp. 123–141.
4. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal property verification. *TACAS '16*, pp. 387–393.
5. Clang. `http://clang.llvm.org`.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS '08*, pp. 337–340.
7. B. Dutertre and L. de Moura. The Yices SMT solver, 2006. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`.
8. Eclipse. `http://www.eclipse.org/`.
9. N. Eén and N. Sörensson. An extensible SAT-solver. *SAT '03*, pp. 502–518.
10. J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.
11. J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. *SEFM '16*, pp. 234–252.
12. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *CGO '04*, pp. 55–88.
13. Mono. `http://www.mono-project.com/`.
14. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, J. Giesl. AProVE: Termination and memory safety of C programs (competition contrib.). *TACAS '15*, pp. 417–419.
15. 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. *Journal of Automated Reasoning*, 58(1):33–65, 2017.