

Proving Termination of Programs Automatically with AProVE*

Jürgen Giesl¹, Marc Brockschmidt², Fabian Emmes¹, Florian Frohn¹,
Carsten Fuhs³, Carsten Otto⁶, Martin Plücker¹, Peter Schneider-Kamp⁴,
Thomas Ströder¹, Stephanie Swiderski⁷, and René Thiemann⁵

¹ RWTH Aachen University, Germany

² Microsoft Research Cambridge, UK

³ University College London, UK

⁴ University of Southern Denmark, Denmark

⁵ University of Innsbruck, Austria

⁶ andrena objects AG, Germany

⁷ Interactive Pioneers GmbH, Germany

Abstract. AProVE is a system for automatic termination and complexity proofs of Java, C, Haskell, Prolog, and term rewrite systems (TRSs). To analyze programs in high-level languages, AProVE automatically converts them to TRSs. Then, a wide range of techniques is employed to prove termination and to infer complexity bounds for the resulting TRSs. The generated proofs can be exported to check their correctness using automatic certifiers. For use in software construction, we present an AProVE plug-in for the popular Eclipse software development environment.

1 Introduction

AProVE (Automated Program Verification Environment) is a tool for automatic termination and complexity analysis. While previous versions (described in [19, 20]) only analyzed termination of term rewriting, the new version of AProVE also analyzes termination of Java, C, Haskell, and Prolog programs. Moreover, it also features techniques for automatic complexity analysis and permits the certification of automatically generated termination proofs. To analyze programs, AProVE uses an approach based on symbolic execution and abstraction [11] to transform the input program into a *symbolic execution graph*¹ that represents all possible computations of the input program. Language-specific features (such as sharing effects of heap operations in Java, pointer arithmetic and memory safety in C, higher-order functions and lazy evaluation in Haskell, or extra-logical predicates in Prolog) are handled when generating this graph. Thus, the exact definition of the graph depends on the considered programming language. For termination or complexity analysis, the graph is transformed into a TRS. The success of AProVE

* Supported by the DFG grant GI 274/6-1 and the FWF grant P22767. Most of the research was done while the authors except R. Thiemann were at RWTH Aachen.

¹ In earlier papers, this was often called a *termination graph*.

at the annual international *Termination Competition* demonstrates that our rewriting-based approach is well suited for termination analysis of real-world programming languages.² A graphical overview of our approach is displayed on the side.³

Technical details on the techniques for transforming programs to TRSs and for analyzing TRSs can be found in [5–9, 15–18, 21–23, 27, 28, 30]. In the current paper, we focus on their implementation in AProVE, which we now made available as a plug-in for the popular Eclipse software development environment [13]. In this way, AProVE can already be applied during program construction (e.g., by analyzing termination of single Java methods for user-specified classes of inputs). In addition to the full version of AProVE, we also made AProVE’s front-ends for the different programming languages available as separate programs. Thus, they can be coupled with other external tools that operate on TRSs, integer transition systems, or symbolic execution graphs. These external tools can then be used as alternative back-ends. Finally, AProVE can also be accessed directly via a web interface [2].

We describe the use of AProVE for the different programming languages and TRSs in Sect. 2. To increase the reliability of the generated proofs, AProVE supports their certification, cf. Sect. 3. We end with a short conclusion in Sect. 4.

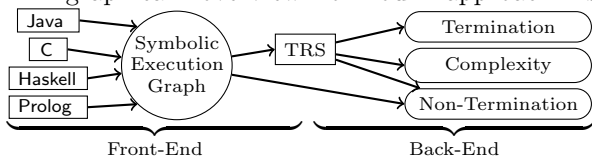
2 AProVE and Its Graphical User Interface in Eclipse

AProVE and its graphical user interface are available as an Eclipse plug-in at [2] under “Download”. After the initial installation, “Check for Updates” in the “Help” menu of Eclipse also checks for updates of AProVE. As Eclipse and AProVE are written in Java, they can be used on most operating systems.

2.1 Analyzing Programming Languages

The screenshot on the next page shows the main features of our AProVE plug-in. Here, AProVE is applied on a Java (resp. Java Bytecode (JBC)) program in the file `List.jar` and tries to prove termination of the `main` method of the class `List`, which in turn calls the method `contains`. (The source code is shown in the editor window (B).) Files in an Eclipse project can be analyzed by right-clicking on the file in Eclipse’s Project Explorer (A) and selecting “Launch AProVE”.⁴

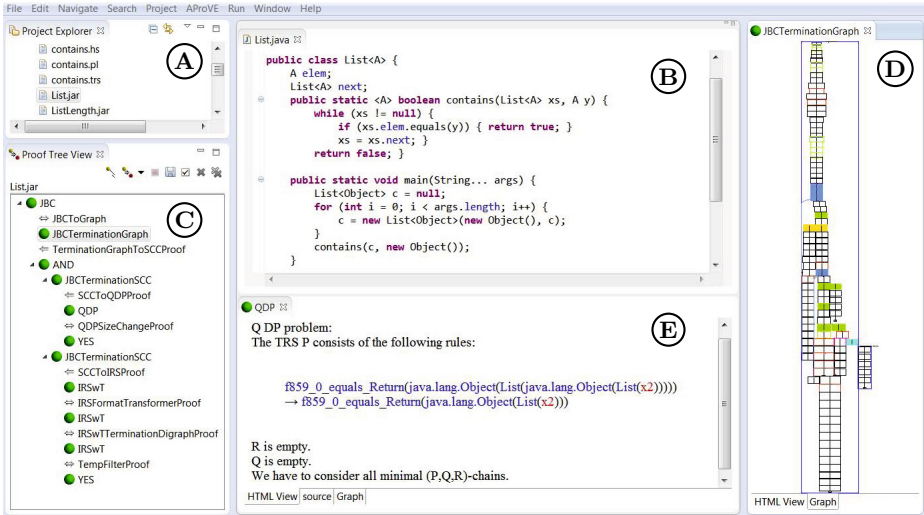
When AProVE is launched, the proof (progress) can be inspected in the Proof Tree View (C). Here, problems (e.g., programs, symbolic execution graphs, TRSs, ...) alternate with proof steps that modify problems, where “ \Leftarrow ” indicates sound



² See http://www.termination-portal.org/wiki/Termination_Competition


³ While termination can be analyzed for Java, C, Haskell, Prolog, and TRSs, the current version of AProVE analyzes complexity only for Prolog and TRSs.

⁴ An initial “ExampleProject” with several examples in different programming languages can be created by clicking on the “AProVE” entry in Eclipse’s menu bar.



and “ \Leftrightarrow ” indicates sound and complete steps. This information is used to propagate information from child nodes to the parent node. A green (resp. red) bullet in front of a problem means that termination of the problem is proved (resp. disproved) and a yellow bullet denotes an unsuccessful (or unfinished) proof. Since the root of the proof tree is always the input problem, the color of its bullet indicates whether AProVE could show its termination resp. non-termination.

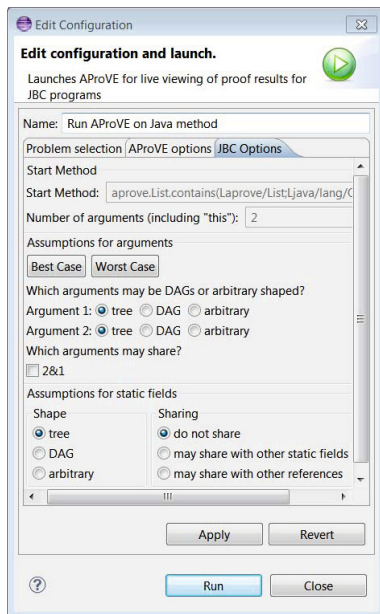
To handle Java-specific features, AProVE first constructs a symbolic execution graph (D) from the program [5–7, 28]. From the cycles of this graph, TRSs are created whose termination implies termination of the original program.⁵ Double-clicking on a problem or proof step in the proof tree shows detailed information about them. For example, the symbolic execution graph can be inspected by double-clicking on the node `JBCTerminationGraph` and selecting the `Graph` tab in the Problem View (D). This graph can be navigated with the mouse, allowing to zoom in on specific nodes or edges. Similarly, one of the generated TRSs is shown in the Problem View (E). For *non-termination* proofs [6], witness executions are provided in the Problem View. In contrast to termination proofs, these analyses are performed directly on the symbolic execution graph.

The buttons in the upper right part of the Proof Tree View (C) interact with AProVE (e.g.,  aborts the analysis). When AProVE is launched, the termination proof is attempted with a time-out of 60 seconds. If it is aborted, one can right-click on a node in the proof tree and by selecting “Run”, one can continue the proof at this node (here, one may also specify a new time-out).

For Java programs, there are two options to specify which parts of the program are analyzed. AProVE can be launched on a `jar` (Java archive) file, and then tries

⁵ These TRSs are represented as *dependency pair problems* [21] (“QDP” in (C)).

to prove termination of the `main` method of the archive’s “main class”.⁶ Alternatively, to use AProVE during software development, single Java methods can be analyzed. Eclipse’s Outline View (reachable via “Window” and “Show View”) shows the methods of a class opened by a double-click in Eclipse’s Project Explorer. An initial “JavaProject” with a class `List` can be created via the “AProVE” entry in Eclipse’s menu bar. Right-clicking on a method in the Outline View and choosing “Launch AProVE” leads to the configuration dialog on the side. It can be used to specify the sharing and shape of the method’s input values. Each argument can be tree-shaped, DAG-shaped, or arbitrary (i.e., possibly cyclic) [7]. Furthermore, one can specify which arguments may be sharing. Similarly, one can provide assumptions about the contents of static fields. There are also two short-cut buttons which lead to the best- and the worst-case assumption. Moreover, under “AProVE options”, one can adjust the desired time-out for the termination proof and under “Problem selection”, one has the option to replace AProVE’s default strategy with alternative user-defined strategies (a general change of AProVE’s strategy is possible via the “AProVE” entry in Eclipse’s main menu).



C [30], Haskell [22], and Prolog [23] are handled similarly. The function, start terms, or queries to be analyzed can be specified in the input file (as in the *Termination Competition*). Otherwise the user is prompted when the analysis starts. For Prolog, AProVE can also infer asymptotic upper bounds on the number of evaluation steps (i.e., unification attempts) and prove determinacy (i.e., that there is at most one solution).

All our programming language front-ends first construct symbolic execution graphs, which are then used to extract the information relevant for termination as a TRS. Thus, analyzing implementations of the same algorithm in different languages leads to very similar TRSs, as AProVE identifies that the reason for termination is always the same. For example, implementations of a `contains` algorithm in different languages all terminate for the same reason on (finite acyclic) lists, since the length of the list decreases in each recursive call or iteration.

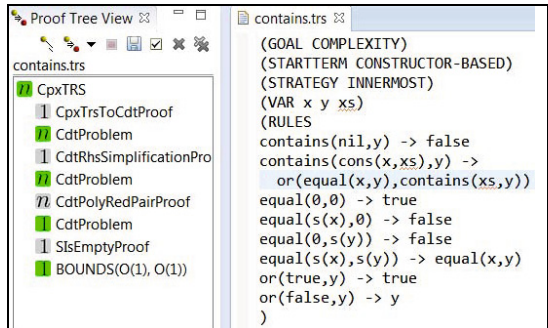
⁶ See http://www.termination-portal.org/wiki/Java_Bytecode for the conventions of the *Termination Competition*, which also specify certain restrictions on the Java programs. In particular, similar to many other termination provers, AProVE treats built-in data types like `int` in Java as unbounded integers \mathbb{Z} . Thus, a termination proof is only valid under the assumption that no overflows occur.

2.2 Analyzing Term Rewrite Systems


To prove termination of TRSs, AProVE implements a combination of numerous techniques within the dependency pair framework [21]. To deal with the pre-defined type of integers in programming languages, AProVE also handles TRSs with built-in integers, using extensions of the dependency pair framework proposed in [16, 18]. To solve the arising search problems (e.g., for well-founded orders), AProVE relies on SAT- and SMT-based techniques like [1, 9, 17, 29]. As SAT solvers, AProVE uses SAT4J [24] and MiniSAT [14]. Like AProVE, SAT4J is implemented in Java and hence, AProVE calls it for small SAT instances, where it is very efficient. MiniSAT is used on larger SAT instances, but as it is invoked as an external process, it leads to a small overhead. As SMT solvers, AProVE uses Yices [12] and Z3 [25]. Non-termination of TRSs is detected by suitable adaptations of narrowing [15].

For complexity analysis, AProVE infers runtime complexity of innermost rewriting. Runtime complexity means that one only considers initial terms $f(t_1, \dots, t_m)$ where t_1, \dots, t_m represent data (thus, they are already in normal form). This corresponds to the setting in program analysis.

Similarly, the analysis of innermost rewriting is motivated by the fact that the transformations from Sect. 2.1 yield TRSs where it suffices to consider innermost rewriting in the back-end. (Polynomial) upper bounds on the runtime complexity are inferred by an adaption of dependency pairs for complexity analysis [27]. To solve the resulting search problems, AProVE re-uses the techniques from termination analysis to generate suitable well-founded orders. As shown in the screenshot, AProVE easily infers that the above TRS has linear asymptotic complexity. More precisely, the **li** at the root node of the proof tree means that initial terms $f(t_1, \dots, t_m)$ of size n only have evaluations of length $\mathcal{O}(n)$.⁷



3 Partial Certification of Generated Proofs

Like any large software product, AProVE had (and very likely still has) bugs. To allow verification of its results, it can export generated termination proofs as machine-readable CPF (Certification Problem Format)⁸ files by clicking on the button  of the Proof Tree View. Independent certifiers can then check the validity of all proof steps. Examples for such certifiers are CeTA [31], CiME/Coccinelle

⁷ Moreover, proof steps also result in complexities (e.g., **li** or **ni**). More precisely, in each proof step, a problem P is transformed into a new problem P' and a complexity c . Then the complexity of P is bounded by the maximum of P' 's complexity and of c .

⁸ See <http://c1-informatik.uibk.ac.at/software/cpf/>

[10], and CoLoR/Rainbow [4]. Their correctness has been formally proved using Isabelle/HOL [26] or Coq [3]. To certify a proof in AProVE’s GUI, one can also call CeTA directly using the button of the Proof Tree View.

Some proof techniques (like the transformation of programming languages to TRSs in AProVE) are not yet formalized in CPF. Until now, proofs with such steps could not be certified at all. As a solution, we extended CPF by an additional element `unknownProof` for proof steps which are not supported by CPF. In the certification, `unknownProof` is treated as an axiom of the form $P_0 \leftarrow P_1 \wedge \dots \wedge P_n$. This allows to prove P_1, \dots, P_n instead of the desired property P_0 . Each P_i can be an arbitrary property such as (non-)termination of some TRS, and P_i ’s subproof can be checked by the certifier again. In this way, it is possible to certify large parts of *every* termination proof generated by AProVE. For example, now 90% of AProVE’s proof steps for termination analysis of the 4367 TRSs in the *termination problem data base (TPDB)*⁹ can be certified by CeTA.

Moreover, we added a new CPF element `unknownInput` for properties that cannot be expressed in CPF, like termination of a Java program. The only applicable proof step to such a property is `unknownProof`. Using `unknownInput`, CPF files for *every* proof can be generated. Now the program transformations in AProVE’s front-end correspond to `unknown` proof steps on `unknown` inputs, but the reasoning in AProVE’s back-end can still be checked by a certifier (i.e., proof steps can transform `unknownInput` into objects that are expressible in CPF).

Due to this new *partial* certification, three bugs of AProVE have been revealed (and fixed) which could be exploited to prove termination of a non-terminating TRS. These bugs had not been discovered before by certification, as the errors occurred when analyzing TRSs resulting from logic programs. If one is only interested in completely certified proofs, the “AProVE” entry in Eclipse’s main menu allows to change AProVE’s default strategy to a “certifiable” strategy which tries to use proof techniques that can be exported to CPF whenever possible.

4 Conclusion

We presented a new version of AProVE to analyze termination of TRSs and programs for four languages from prevailing programming paradigms. Moreover, AProVE analyzes the runtime complexity of Prolog programs and TRSs. We are currently working on extending AProVE’s complexity analysis to Java as well [8].

AProVE’s power is demonstrated by its performance in the annual *Termination Competition*, where it won almost all categories related to termination of Java, Haskell, Prolog, and to termination or innermost runtime complexity of TRSs. Moreover, AProVE participated very successfully in the *SV-COMP* competition¹⁰ at *TACAS* which featured a category for termination of C programs for the first time in 2014. AProVE’s automatically generated termination proofs can be exported to (partially) check them by automatic certifiers. Our tool is available as a plug-in of the well-known Eclipse software development environment.

⁹ The *TPDB* is the collection of examples used in the annual *Termination Competition*.

¹⁰ See <http://sv-comp.sosy-lab.org/2014/>

Moreover, the front-ends of AProVE for the different programming languages are also available separately in order to couple them with alternative back-ends. To download AProVE or to access it via a web interface, we refer to [2].

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
2. AProVE, <http://aprove.informatik.rwth-aachen.de/>
3. Bertot, Y., Castéran, P.: Coq’Art. Springer (2004)
4. Blanqui, F., Koprowski, A.: CoLoR: A Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science* 4, 827–859 (2011)
5. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java Bytecode programs by term rewriting. In: Schmidt-Schauß, M. (ed.) RTA 2011. LIPIcs, vol. 10, pp. 155–170. Dagstuhl Publishing (2011)
6. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 123–141. Springer, Heidelberg (2012)
7. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 105–122. Springer, Heidelberg (2012)
8. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating runtime and size complexity analysis of integer programs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 140–155. Springer, Heidelberg (2014)
9. Codish, M., Giesl, J., Schneider-Kamp, P., Thiemann, R.: SAT solving for termination proofs with recursive path orders and DPs. *JAR* 49(1), 53–93 (2012)
10. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Automated certified proofs with CiME3. In: Schmidt-Schauß, M. (ed.) RTA 2011. LIPIcs, vol. 10, pp. 21–30. Dagstuhl Publishing (2011)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL 1977, pp. 238–252. ACM Press (1977)
12. Dutertre, B., de Moura, L.M.: The Yices SMT solver (2006), tool paper at <http://yices.csl.sri.com/tool-paper.pdf>
13. Eclipse, <http://www.eclipse.org/>
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Emmes, F., Enger, T., Giesl, J.: Proving non-looping non-termination automatically. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 225–240. Springer, Heidelberg (2012)
16. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) RTA 2011. LIPIcs, vol. 10, pp. 41–50. Dagstuhl Publishing (2011)

17. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
18. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
19. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)
20. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
21. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
22. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. TOPLAS 33(2), 7:1–7:39 (2011)
23. Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., Fuhs, C.: Symbolic evaluation graphs and term rewriting — A general methodology for analyzing logic programs. In: De Schreye, D., Janssens, G., King, A. (eds.) PPDP 2012, pp. 1–12. ACM Press (2012)
24. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2. JSAT 7, 59–64 (2010)
25. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
26. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
27. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. JAR 51(1), 27–56 (2013)
28. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java Bytecode by term rewriting. In: Lynch, C. (ed.) RTA 2010. LIPIcs, vol. 6, pp. 259–276. Dagstuhl Publishing (2010)
29. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
30. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P.: Proving termination and memory safety for programs with pointer arithmetic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 204–218. Springer, Heidelberg (2014)
31. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)