Terminator: Beyond Safety
(Tool Paper)

Byron Cook[1], Andreas Podelski[2,3], and Andrey Rybalchenko[2,4]

[1] Microsoft Research
[2] Max-Planck-Institut für Informatik
[3] Universität Freiburg, Institut für Informatik
[4] EPFL

Abstract. Previous symbolic software model checkers (*i.e.*, program analysis tools based on predicate abstraction, pushdown model checking and iterative counterexample-guided abstraction refinement, etc.) are restricted to safety properties. Terminator is the first software model checker for termination. It is now being used to prove that device driver dispatch routines always return to their caller (or return counterexamples if they if they fail to terminate).

1 Introduction

Terminator is a program analysis and verification tool for termination. It supports large program fragments (*i.e.*, >20,000 LOC) together with C programming language features such as arbitrarily nested loops, arbitrarily nested recursive functions, pointer-aliasing and side-effects, function-pointers, etc. It is fully automatic; no annotations or auxiliary proof arguments (*e.g.*, ranking functions) need to be provided. It automatically synthesizes the termination argument. In the case where the proof cannot be refined Terminator produces counterexamples in the form of (possibly nested) looping paths through the control flow graph. In program analysis terms, Terminator is interprocedural, path sensitive and context-sensitive. Technically it is based on predicate abstraction, pushdown model checking and iterative counterexample-guided abstraction refinement, *i.e.*, on the ingredients of software model checkers such as Blast [12], Magic [3], Slam [1].

We have applied Terminator to device drivers ranging in sizes from 5,000 to 35,000 LOC in order to prove that their dispatch routines always return to the operating system when called. These experiments were carried out using an integration of Terminator and the Windows Static Driver Verifier[1,15] product. Overall, 8 termination bugs were found in 23 device drivers. The runtime ranged from 5 seconds to 44 hours. A full account of the results can be found in [8]. See also the Terminator home page `http://research.microsoft.com/TERMINATOR`.

2 Termination Analysis for Software

Reactive systems such as operating systems, web servers, mail servers, and database engines are constructed from sets of components that we expect will

always terminate. Cases where these functions unexpectedly do not return to their calling contexts leads to non-responsive systems and system crashes. Proving that these system components always terminate has been a challenge because, until now, no termination tool has ever been able to provide the necessary capacity (>20,000 LOC) together with accurate support for programming language features such as arbitrarily nested loops and recursive functions, pointers and side-effects, function-pointers, etc. TERMINATOR fills this gap.

In the context of program analysis and model checking, tools checking programs over infinite data spaces have been targeted at safety properties. These tools are usually based on abstraction. While the preservation of termination properties from the abstract to the concrete system is sound (if the abstract system terminates then so does the concrete one), it is also worthless in all but pathological cases (with classical abstraction techniques related to homomorphic abstraction or simulation, the abstraction to a finite graph will 'always' introduce a loop and thus it will *not* preserve the termination property).

There exist tools for proving termination for very specialized classes of programs and calculi, such as rewriting [11], logic and functional programming [5,13,14], and imperative programs with specific arithmetic operations [2,6,9]. None of these tools targets scalability and the features of practical programming languages.

3   Foundations Behind Terminator

TERMINATOR is the culmination of successive research, namely (i) a new proof rule for termination, (ii) an appropriate form of abstraction for the automation of the proof rule via abstract interpretation, (iii) a form of iterative counterexample-guided refinement not only of the abstraction but also of the candidate termination argument, and finally (iv) a practical algorithm for binary reachability analysis that is used for validation of candidate termination arguments. Below we highlight the theoretical foundations of TERMINATOR.

(i) Termination argument. The termination argument constructed by TERMINATOR is a union of well-founded relations that forms a transition invariant, *i.e.*, a binary relation over program states that contains the transitive closure of the transition relation of the program [17]. One distinguishing feature of transition invariants is that they can be constructed by abstract fixpoint computation. This fits well into the framework of abstract interpretation [10] and thus leads naturally to automatic methods.

(ii) Abstraction for termination. Transition predicate abstraction [18] overcomes the above-mentioned limitation of classical abstraction techniques to the verification of safety properties. Transition predicate abstraction induces a finite graph where nodes are labeled by abstract transitions. Termination is determined by the well-foundedness of those abstract transitions, and not the absence of loops. Transition predicate abstraction can be refined in the classical way, namely by adding more predicates [4].

(iii) Refinement for termination. As described in [7], TERMINATOR incrementally constructs a candidate transition invariant and thus iteratively *refines* the termination argument. This refinement is again guided by counterexamples. A counterexample is here a path that leads some state s to some state s' such that the pair (s, s') violates the candidate transition invariant (which does not yet fully contain the transitive closure of the transition relation). The path, a sequence of statements, may be viewed as a program. TERMINATOR uses the ranking function synthesis tool RANKFINDER [16] to compute a ranking function for this program. The corresponding ranking relation consists of all pairs of states with decreasing rank, including the pair (s, s'). The refinement of the termination argument amounts to adding this relation to the candidate transition invariant (a union of well-founded relations).

(iv) Binary reachability analysis. In the refinement loop described above, for each new candidate transition invariant, we need to check its validity (the fact that it contains the transitive closure of the transition relation). TERMINATOR implements this check, the *binary* reachability analysis, using a second kind of refinement, namely counterexample-guided *abstraction* refinement (viz. of transition predicate abstraction). In contrast, a safety property translates to one fixed invariant, whose validity is checked by (standard, *unary*) reachability analysis. The crux of TERMINATOR's implementation of binary reachability analysis is to reduce each new binary reachability problem to a (unary) reachability problem for a new program constructed by a syntactic transformation from the given program [8]. After each transformation, TERMINATOR applies (unary) reachability analysis. In a sense, TERMINATOR implements a reduction of termination not to a safety property but to the *existence* of a certain safety property.

4 Beyond Termination

In some cases, termination depends on additional properties (such as: the repeated request will eventually be served) that can be modeled as *fairness* assumptions. Termination is an example of a basic *liveness* property. We are working on the next generation of TERMINATOR that will prove general liveness properties under fairness assumptions.

References

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys'06: European Systems Conference*, pages 73–85, 2006.
2. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VM-CAI'05: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 113–129. Springer, 2005.
3. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03: International Conference on Software Engineering*, pages 385–395. IEEE, 2003.

4. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00: Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

6. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'02: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer, 2002.

7. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS'05: Static Analysis Symposium*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

8. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation (to appear)*, 2006.

9. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.

10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79: Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

11. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA'04: Rewriting Techniques and Applications*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

12. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL'04: Principles of Programming Languages*, pages 232–244. ACM Press, 2004.

13. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'01: Principles of Programming Languages*, volume 36, 3 of *ACM SIGPLAN Notices*, pages 81–92. ACM Press, 2001.

14. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In *CAV'97: Computer-Aided Verification*, LNCS, pages 444–447. Springer, 1997.

15. Microsoft Corporation. Windows Static Driver Verifier. Available at www.microsoft.com/whdc/devtools/tools/SDV.mspx, July 2004.

16. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04: Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.

17. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'04: Logic in Computer Science*, pages 32–41. IEEE, 2004.

18. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'05: Principles of Programming Languages*, pages 132–144. ACM Press, 2005.