Termination Analysis with Compositional Transition Invariants*

Daniel Kroening[1], Natasha Sharygina[2,4],
Aliaksei Tsitovich[2], and Christoph M. Wintersteiger[3]

[1] Oxford University, Computing Laboratory, UK
[2] Formal Verification and Security Group, University of Lugano, Switzerland
[3] Computer Systems Institute, ETH Zurich, Switzerland
[4] School of Computer Science, Carnegie Mellon University, USA

Abstract. Modern termination provers rely on a safety checker to construct disjunctively well-founded transition invariants. This safety check is known to be the bottleneck of the procedure. We present an alternative algorithm that uses a light-weight check based on transitivity of ranking relations to prove program termination. We provide an experimental evaluation over a set of 87 Windows drivers, and demonstrate that our algorithm is often able to conclude termination by examining only a small fraction of the program. As a consequence, our algorithm is able to outperform known approaches by multiple orders of magnitude.

1 Introduction

Automated termination analysis of systems code has advanced to a level that permits industrial application of termination provers. One possible way to obtain a formal argument for termination of a program is to *rank* all states of the program with natural numbers such that for any pair of consecutive states s_i, s_{i+1} the rank is decreasing, i.e., $rank(s_{i+1}) < rank(s_i)$. In other words, a program is terminating if there exists a *ranking function* for every program execution.

Substantial progress towards the applicability of procedures that compute ranking arguments to industrial code was achieved by an algorithm called *Binary Reachability Analysis (BRA)*, proposed by Cook, Podelski, and Rybalchenko [1]. This approach combines detection of ranking functions for program paths with a procedure for checking safety properties, e.g., a Model Checker. The key idea of the algorithm is to encode an intermediate termination argument into a program annotated with an assertion, which is then passed to the safety checker. Any counterexample for the assertion produced by the safety checker contains a path that violates the intermediate termination argument. The counterexample path is then used to compute a better termination argument with the help of methods that discover ranking functions for program paths.

* Supported by the Swiss National Science Foundation under grant no. 200020-122077, by EPSRC grant no. EP/G026254/1, by the EU FP7 STREP MOGENTES, and by the Tasso Foundation.

A broad range of experiments with different implementations have shown that the bottleneck of this approach is the safety check [1, 2]: Cook et al. [1] report more than 30 hours of runtime for some of their benchmarks. The time for computing the ranking function for a given program path is insignificant in comparison. Part of the reason for the difficulty of the safety checks is their dual role: they ensure that a disjunctively composed termination argument is correct and they need to provide sufficiently deep counterexamples for the generation of further ranking arguments.

We propose a new algorithm for termination analysis that addresses these challenges as follows: 1) We use a light-weight criterion for termination based on *compositionality* of transition invariants. 2) Instead of using full counterexample paths, the algorithm applies the path ranking procedure directly to increasingly deep unwindings of the program until a suitable ranking argument is found. We prove soundness and completeness (for finite-state programs) of our approach and support it by an extensive evaluation on a large set of Windows device drivers. Our algorithm performs up to 3 orders of magnitude faster than BRA, as it avoids the bottleneck of safety checking in the iterative construction of a termination argument.

2 Background

Preliminaries We define notation for programs and record some basic properties we require later on. Programs are modeled as *transition systems*.

Definition 1 (Transition System). *A transition system (program) P is a three tuple $\langle S, I, R\rangle$, where*

- *S is a (possibly infinite) set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is the transition relation.*

A *computation* of a transition system is a maximal sequence of states s_0, s_1, ... such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. A program is terminating iff all computations of the program eventually reach a final state. The non-reflexive transitive closure of R is denoted by R^+, and the reflexive transitive closure of R is denoted by R^*. The set of reachable states is $R^*(I)$.

Podelski and Rybalchenko [3] use *Transition Invariants* to prove termination of programs:

Definition 2 (Transition Invariant [3]). *A* transition invariant *T for program $P = \langle S, I, R\rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.*

A well-founded relation is a relation that does not contain infinite descending chains. Podelski and Rybalchenko define a weaker notion as follows:

Definition 3 (Disjunctive Well-foundedness [3]). *A relation T is* disjunctively well-founded (d.wf.) *if it is a finite union $T = T_1 \cup \ldots \cup T_n$ of well-founded (wf.) relations.*

A program is terminating if it does not have infinite computations, and Podelski and Rybalchenko show that disjunctive well-foundedness is enough to prove termination of a program:

Theorem 1 (Termination [3]). *A program P is terminating iff there exists a d.wf. transition invariant for P.*

The literature presents a broad range of methods to obtain transition invariants. Usually, this is accomplished via synthesis of *ranking functions*, which define well-founded *ranking relations* [2, 4–6]. We refer to such methods as *ranking procedures*.

Binary Reachability Analysis [1]. Theorem 1 gives rise to an algorithm for proving termination that constructs a d.wf. transition invariant in an incremental fashion. Initially, an empty termination argument is used, i.e., $T_0 = \emptyset$. Then, a Model Checker is used to search the reachable state space for a counterexample to termination argument T_i. If there is none, termination is proven. Otherwise, let π be the counterexample path. The counterexample may be genuine, i.e., demonstrate a prefix of a non-terminating computation. Otherwise, a well-founded relation T that includes π is constructed (via a ranking procedure). Finally, the current termination argument is updated, i.e., $T_{i+1} = T_i \cup T$ and the process is repeated.

This principle has been put to the test in various tools, most notably in Terminator [1], ARMC [7], and in an experimental version of SatAbs [2].

3 Compositional Termination Analysis

The literature contains a broad range of reports of experiments with multiple implementations that indicate that the bottleneck of Binary Reachability Analysis is that the safety checks are often difficult to decide by means of the currently available software Model Checkers [1, 2]. This problem unfortunately applies to both cases of finding a counterexample to an intermediate termination argument and to proving that no such counterexample exists.

As an example, consider a program that contains a trivial loop. The d.wf. transition invariant for the loop can be constructed in a negligible amount of time, but the computation of a path to the beginning of the loop may already exceed the computational resources available.

In this section, we describe a new algorithm for proving program termination that achieves the same result while avoiding excessively expensive safety checks.

We first define the usual relational composition operator $\circ$ for two relations $A, B : S \times S$ as

$$A \circ B := \{(s, s') \mid \exists s''.(s, s'') \in A \wedge (s'', s') \in B\} \ .$$

Note that a relation R is transitive if it is closed under relational composition, i.e., when $R \circ R \subseteq R$. To simplify presentation, we also define $R^1 := R$ and $R^n := R^{n-1} \circ R$ for any relation $R : S \times S$.

While d.wf. transition invariants are not in general well-founded, there is a trivial subclass for which this is the case:

Definition 4 (Compositional Transition Invariant). *A d.wf. transition invariant T is called* compositional *if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.*[5]

A compositional transition invariant is of course well-founded, since it is an inductive transition invariant for itself [3]. Using this observation and Theorem 1, we conclude:

Corollary 1. *A program P terminates if there exists a compositional transition invariant for P.*

In Binary Reachability Analysis, the Model Checker needs to compute a counterexample to an intermediate termination argument, which is often difficult. The counterexample begins with a *stem*, i.e., a path to the entry point of the loop. For many programs, the existence of a d.wf. transition invariant does not actually depend on the entry state of the loop. For example, termination of a trivial loop that increments a variable i to a given upper limit u does not actually depend on the initial value of i, nor does it depend on u. The assurance of progress towards u is enough to conclude termination.

The other purpose of the Model Checker in BRA is to check that a candidate transition invariant indeed includes R^+ restricted to the reachable states. To this end, we note that the (non-reflexive) transitive closure of R is essentially an unwinding of program loops:

$$R^+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{i=1}^{\infty} R^i \ .$$

Instead of searching for a d.wf. transition invariant that is a superset of R^+, we can therefore decompose the problem into a series of smaller ones. We consider a series of loop-free programs in which R is unwound k times, i.e., the program that contains the transitions in $R^1 \cup \ldots \cup R^k$.

Observation 2. *Let $P = \langle S, I, R \rangle$ and $k \geq 1$. If there is a d.wf. T_k with $\bigcup_{j=1}^{k} R^j \subseteq T_k$ and T_k is also transitive, then T_k is a compositional transition invariant for P.*

Proof. We show that T_k is a transition invariant for P, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T_k$. Let $(x, x') \in R^+ \cap (R^*(I) \times R^*(I))$. There must exist a path over R-edges from x to x'. Let l be the length of the path, i.e., $(x, x') \in R^l$. Note that $R \subseteq T_k$, and thus, $R^l \subseteq T_k^l$. As T_k is transitive, $T_k^l \subseteq T_k$. □

This suggests a trivial algorithm that attempts to construct d.wf. relations T_i for incrementally deep unwindings of P until it finally finds a transitive T_k, which

[5] We use the term *compositional* instead of *transitive* for transition invariants in order to comply with the terminology in the existing literature [3].

proves termination of P. However, this trivial algorithm need not terminate, even for simple inputs. This is due to the fact that T_i does not necessarily have to be different from T_{i-1}, in which case the algorithm will never find a compositional transition invariant.

We provide an alternative that does not suffer from this limitation and takes advantage of the fact that most terminating loops encountered in practice have transition invariants with few disjuncts. To present this algorithm, we require the following lemma, which enables us to exclude computations from the program that we have already proven terminating in a previous iteration:

Lemma 1. *Let $P = \langle S, I, R\rangle$ and $k \geq 1$. Let $T_1, \ldots, T_k$ be a sequence of d.wf. relations such that each is a superset of the respective $\bigcup_{j=1}^{i} R^j$ restricted to reachable transitions that are not contained in any previous T_j, i.e.,*

$$\bigcup_{j=1}^{i} R^j \setminus \bigcup_{j=1}^{i-1} T_j \cap (R^*(I) \times R^*(I)) \subseteq T_i \ .$$

If $Q := \bigcup_{i=1}^{k} T_i$ is transitive, then Q is a compositional transition invariant for the program P.

Proof. We have $\bigcup_{i=1}^{k} R^i \subseteq \bigcup_{i=1}^{k} T_i = Q$ and in particular $R \subseteq Q$. Therefore $R^+ \subseteq Q^+$ and since Q is transitive it follows that $R^+ \subseteq Q$. It is d.wf. as it is a finite union of d.wf. relations. □

As an optimization, we may safely omit some of the T_i while searching for a transitive T_k:

Lemma 2 (Optimization). *Let $T_0, \ldots, T_k$ be the sequence of d.wf. relations for application of Lemma 1. The claim of the lemma holds even if some of the $T_1, \ldots, T_{k-1}$ are not provided (empty).*

Proof. We show that Q is a transition invariant for P. Let $(x, x') \in R^+ \cap (R^*(I) \times R^*(I))$. As in the proof of Obs. 2, $(x, x') \in R^l$ for some l. The claim holds trivially for $l \leq k$ as $\bigcup_{i=1}^{k} R^i \subseteq Q$. For $l > k$, note that $(x, x') \in (R^{jk} \circ R^{l-jk})$ and $0 \leq l - jk < k$ for some $j \geq 1$. Note that $R^{jk} \subseteq Q^j$ and $R^{l-jk} \subseteq Q$. Thus, $(x, x') \in (Q^j \circ Q) = Q^{j+1}$. As Q is transitive, $Q^{j+1} \subseteq Q$, and thus $(x, x') \in Q$.

The proof of Lemma 1 still applies. As an example, our implemenation only uses those T_i where i is a power of two.

The procedure that we draw from Lemma 2 is Algorithm 1, and we call it *Compositional Termination Analysis* (CTA). This algorithm makes use of an external ranking procedure called *rank*, which generates a d.wf. ranking relation for a given set of transitions, or alternatively a set $C \in S$ of states such that $R^*(C)$ contains infinite computations. We say that *rank* is sound if it always returns either a d.wf. superset of its input or a non-empty set of states C, and we call it complete if it terminates on every input.

```
   input  : P = ⟨S, I, R⟩
   output : 'Terminating' / 'Non-Terminating'
 1 begin
 2     T := ∅;
 3     X := S;
 4     i := 1;
 5     while true do
 6         ⟨T_i, C⟩ := rank ((⋃_{j=1}^{i} R^j \ T) ∩ (X × X));
 7         if C ∩ R*(I) ≠ ∅ then
 8             return 'Non-Terminating';
 9         else if C = ∅ and T ∪ T_i is transitive then
10             return 'Terminating';
11         else
12             X := X \ C;
13             T := T ∪ T_i;
14             i := i + j, where j > 0;
15         end
16     end
17 end
```

Algorithm 1: Compositional Termination Analysis

Algorithm 1 maintains a set $X \subseteq S$ that is an over-approximation of the set of reachable states, i.e., $R^*(I) \subseteq X$. It starts with $X = S$ and at $i = 1$. It iterates over i and generates d.wf. ranking relations T_i for the transitions in $\bigcup_{j=1}^{i} R^j \setminus T$. As long as such relations are found, they are added to T. Once it finds a transitive T, the algorithm stops, as P terminates according to Lemma 2. When ranking fails for some i, the algorithm checks whether there is a reachable state in C, in which case $R^*(C)$ contains a counterexample to termination and the algorithm consequently reports P as non-terminating. Otherwise, it removes C from X, which represents a refinement of the current over-approximation of the set of reachable states.

Theorem 3. *Assuming the sub-procedure rank is sound, Algorithm 1 is sound.*

Proof. When the algorithm terminates with 'terminating' (line 10), the sequence of relations T_i constructed so far is suitable for application of Lemma 2, which proves termination. It is easy to see that the set $R^*(I)$ in Lemma 2 can be over-approximated to X. If the algorithm returns 'non-terminating' at line 8, it has found a set of reachable states from which infinite computations exist, i.e., there is a concrete counterexample to termination. □

Lines 12–14 ensure progress between iterations by excluding unreachable states (C) from the approximation X and adding the most recently found T_i in T. However, for non-terminating input programs, the algorithm may not terminate for two reasons: a) *rank* is not required to terminate, and b) there may be an infinite sequence of iterations. This is not the case for finite S if the input

program is non-terminating, since sound and complete ranking procedures exist (e.g., [5, 2]) and progress towards the goal can thus be ensured:

Corollary 2. *If the sub-procedure rank is sound and complete for finite-state programs, then Algorithm 1 is sound and complete for non-terminating finite-state programs.*

Proof. We assume a non-terminating input program $P = \langle S, I, R \rangle$. As S is finite there must exist a looping counterexample with a finite stem. In each iteration, either T increases or X decreases, as $C \cap X = \emptyset$. Thus, the algorithm will eventually consider an unwinding long enough to contain the stem, at which point *rank* returns a C with $C \cap R^*(I) \neq \emptyset$ (since it is sound and complete). In both cases, progress is ensured because *rank* always returns a d.wf. ranking relation or a non-empty set C. In the worst case, the number of iterations is equal to the length of the shortest counterexample to termination. □

Note that the algorithm is not complete for terminating programs even if they are finite-state. This is due to the fact that T is not guranteed to ever become transitive, even if it contains R^+.

Example. We demonstrate the advantage of our approach over BRA on the following simple program, where $*$ represents non-deterministic choice.

```
integer i;

while i<255 do begin
  if * then i:=i+1;
  else i:=i+2;
end
```

The state space in this example is $S = \mathbb{N}_0$, and i is the only variable. A suitable wf. transition invariant is $\{(i, i') \in S^2 \mid i < i' \wedge i' \leq 256\}$, which is easily generated within a negligible amount of time. BRA subsequently needs to verify the absence of further counterexamples, which requires 14 refinement iterations when the SATABS engine is used. Compositional Termination Analysis returns immediately after synthesizing the ranking function, because the corresponding relation is transitive. A different ranking procedure may return a d.wf. transition invariant with one disjunct for each path through the loop body. In this case, our algorithm stops in the second iteration, because there are no more transitions that are not included in either of the two disjuncts.

Remark. The check in line 9 of the algorithm corresponds to checking whether $T \circ T \subseteq T$ for some relation $T : S \times S$. This corresponds to checking validity of

$$\forall x, y \in S.\ (x, y) \in T \rightarrow (x, y) \in T \circ T\ ,$$

which, in the case of symbolically-represented relations, can be established using one call to a suitable decision procedure.

4 Implementation

We have implemented Compositional Termination Analysis for ANSI-C programs. Our implementation follows Algorithm 1. It instruments the program with termination assertions as described by Cook et al. [1] and subsequently applies the termination analysis once to each loop in the program. There are two additional features that need discussion, namely our abstracting loop slicer, and the blockwise ranking procedure.

4.1 Slicing and Loop Abstraction

To reduce the resource requirements of the Model Checker, our implementation analyzes each loop separately. It generates an inter-procedural slice [8] of the program, slicing backwards from the termination assertion. In addition, we rewrite the program into a single-loop program, abstracting from the behavior of all other loops.

Following the hypothesis that loop termination rarely depends on complex variables that are possibly calculated by other loops, our slicing algorithm replaces all assignments that depend on five or more variables with non-deterministic values. Also all loops other than the analyzed one are 'havocked': they are replaced by program fragments that assign non-deterministic values to all variables that might change during the execution of the loop (similar to the loop summarization in [9]).

Note that this is a purely practical issue: The benchmarks we use require far too much time to run without this abstraction. We have noticed however, that the abstraction is almost always precise enough, i.e., we loose only very few termination proofs. Of course, we use the exactly same slices for all methods that we compare in our evaluation.

4.2 Blockwise Ranking

The sub-procedure *rank* in Algorithm 1 may be implemented in various ways. For example, it is possible to enumerate all paths through $\bigcup_{j=0}^{i} R^j$ and to obtain a d.wf. ranking relation for every path separately. To avoid this enumeration, we employ the symbolic execution engine of CBMC [10] to find paths through the program that are not yet included in the candidate transition invariant. For this purpose, we create a temporary program that first initializes all variables with non-deterministic values, saves the state, and then executes R^i, which is loop-free. Finally, we check for inclusion of the loop pre- and post-states in the current candidate transition invariant (starting with the empty set).

If a counterexample is found, we extract a path from it and try to compute a wf. ranking relation for it. If this succeeds, this relation is added disjunctively to the current (d.wf.) candidate transition invariant. This procedure is equivalent to the application of Binary Reachability Analysis to a loop-free program fragment.

The explicit check for compositionality of the candidate transition invariant can often be avoided. If we find that T is composed of a single wf. ranking

```
1  void main()
2  {
3   int x;
4   int debug = 0;
5
6   while(x<255) {
7    if (x%2!=0) x--;
8    else x+=2;
9    if (debug!=0) x=0;
10  }
11 }
```

Fig. 1. A loop with four paths through its body

relation, we trivially know that the transition relation is also well-founded, since it is a subset of a wf. transition invariant.

The synthesis of ranking relations for paths is out of the scope of this paper, but we would like to point the interested reader to some recent results in this area [2, 4, 5].

4.3 Illustration

To illustrate Compositional Termination Analysis and our implementation, we demonstrate the most important steps on a simple program. The source code for this demonstration is given in Figure 1 and it consists of an ANSI-C program that contains a single loop with four paths through its body. Figure 2 shows the control flow graph of this program and defines the program locations l_1 to l_9.

Our algorithm starts at $i = 1$ and R^1 is equivalent to a single unwinding of the loop, i.e., a single copy of the loop body. The initial value of X is S, which allows any entry state of the loop, including states that have the variable *debug* set to values other than 0. The initial termination argument T is $\emptyset$. The procedure *rank* analyzes R^1 and, since T is empty, any path between the locations l_2 and l_8 violates the current termination argument. Consider the path passing through locations $l_2, l_3, l_4, l_6, l_7, l_8$. There is no wf. ranking relation for this path because the segment between locations l_7 and l_8 sets x to 0. This means that x is always set to the same value, which also happens to satisfy the loop entry condition. Furthermore, the variable *debug* never changes its value. Thus, the procedure *rank* returns a non-empty path precondition $C \equiv (\mathit{debug} \neq 0) \wedge (x < 255) \wedge (x\%2 \neq 0)$. However, C does not contain any reachable loop entry state in the original program because *debug* is set to 0 between l_1 and l_2. Consequently, the test at line 7 of Algorithm 1 fails and X is updated to $X \setminus C$ at line 12 of Algorithm 1.

The algorithm continues with a refined X, while T is still empty. There exist two more paths through the block R^1: l_2, l_3, l_4, l_6, l_8 and l_2, l_3, l_5, l_6, l_8. The procedure *rank* finds a ranking function for each of them, namely $+x$ for the first path and $-x$ for the second, and constructs the d.wf. ranking relation $T_1 \equiv$

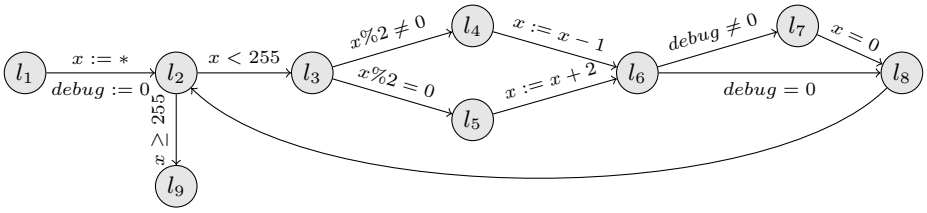

Fig. 2. Control-flow graph of the program in Fig. 1

$x < x' \vee -x < -x'$, which is disjuncively composed of two ranking relations over the pre- and post-state of the loop (x and x', respectively).

The constructed d.wf. ranking relation T_1 is added to the termination argument T and i is increased. Since the d.wf. ranking relation found in the previous iteration was disjunctive, the algorithm proceeds to the next iteration, where *rank* examines R^2, i.e., it now explores two loop unwindings. However, it cannot find any new path that is both in R^2 and not included in T. Therefore, the algorithm concludes that the program in Fig. 1 terminates according to Lemma 2.

5 Experimental Results

We have evaluated our implementation of Compositional Termination Analysis on a set of 87 Windows device drivers taken from the Windows Device Driver Kit.[6] Every driver is analyzed in two different configurations, which results in a total of 174 benchmarks. We use Goto-CC[7] to extract control flow graphs from the original sources, which are then passed to our Compositional Termination Analysis engine.

We compare our implementation to an implementation of Binary Reachability Analysis using SatAbs as the safety checker. In all our experiments, we use a simple and incomplete coefficient enumeration approach to synthesize polynomial ranking functions using a SAT solver. This and our implementation of Binary Reachability Analysis have been used in a recent comparison of ranking engines [2]. For our evaluation we run Binary Reachability Analysis on every driver using a timeout of 2 hours and a memory limit of 2 GB on an Intel Xeon 3 GHz machine. All those loops that this engine analyzes successfully within the time limit serve as the baseline for our comparison. Note that some loops may not require calls to a ranking engine, either because they are unreachable or termination is trivial and shown by preprocessing. We have excluded those loops, i.e., our evaluation is only on loops that require a ranking engine at least once. Our baseline consists of 99 terminating and 45 non-terminating benchmarks.

Whenever the ranking engine is not able to find a valid d.wf. transition invariant for a block, it returns the weakest precondition of a corresponding

[6] Version 6, available online at http://www.microsoft.com/whdc/devtools/wdk/
[7] http://www.cprover.org/goto-cc/

path. This precondition describes a set of states from which termination of the program is not guaranteed. Our implementation can be configured to react to this situation in three different ways: a) check reachability of the precondition, b) check reachability of the loop, or c) report the loop as non-terminating. We present results for all three variants.

First, we discuss the results obtained from variant a), which checks path preconditions using a Model Checker (SATABS in our implementation) and thus features the same level of precision as Binary Reachability Analysis. Every data point in Figure 3 represents one loop. On the horizontal axis we indicate the total time taken to analyze this loop using Binary Reachability Analysis. The vertical axis indicates the time taken by Compositional Termination Analysis. As apparent in Figure 3, Compositional Termination Analysis is up to three orders of magnitude faster. The average speedup factor is 52. However, there are a few non-terminating benchmarks on which it is slower. This is due to the fact that on non-terminating loops many (or all) path preconditions are eventually enumerated. The resulting loop-free programs are sometimes too difficult for the model checker. A possible solution for this problem are techniques that compute a more general precondition of non-termination. A recent technique described by Cook et al. [11], which constructs preconditions of termination, could be applied for this purpose.

Figure 4 provides the results obtained when checking for general loop reachability, which is essentially a crude over-approximation of the precondition of the non-terminating paths through the loop. The results are very similar to those of the previous variant. This is due to the fact that most loops are indeed reachable and so are most path preconditions. There is no difference in precision compared to variant a) on these benchmarks, i.e., no termination proofs are lost compared to an actual precondition check.

Finally, we discuss the results obtained with variant c), which reports non-termination immediately, i.e., without checking reachability of the loop or a precondition. Naturally, this version of our algorithm is the fastest (Fig. 5). The imprecision introduced by not checking loop reachability or path preconditions does not have any effect on these benchmarks.

Figure 6 shows that the overall capacity of Compositional Termination Analysis is much higher than that of Binary Reachability Analysis: At an equal level of precision, Compositional Termination Analysis is able to analyze more than three times the number of benchmarks that Binary Reachability Analysis is able to analyze.

All our experimental data, the implementation, and additional material is available for further research at http://www.cprover.org/termination/.

6 Related work

Termination analysis has its roots in the work of Turing [12, 13]. Since then, substantial progress has been made in various areas of computer science: logic

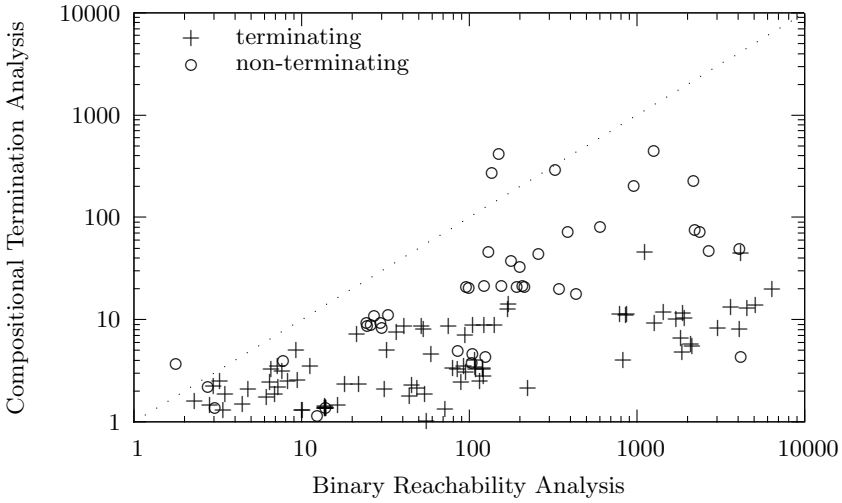

Fig. 3. Experimental results using path precondition checks.

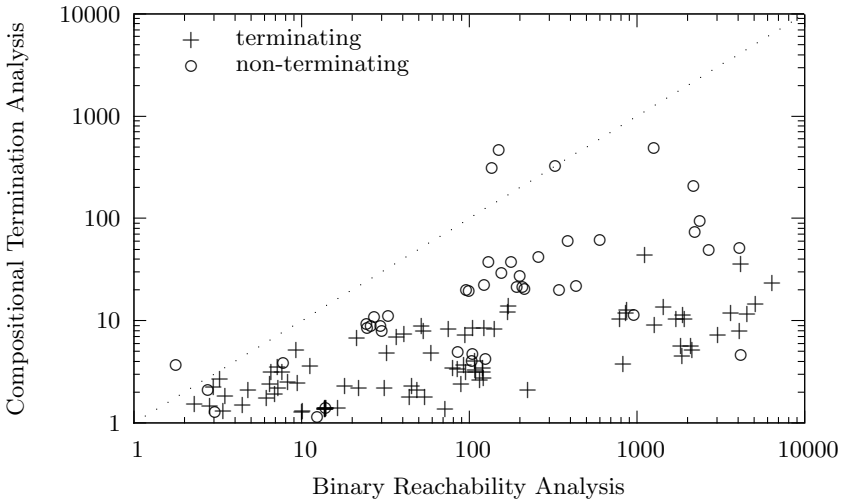

Fig. 4. Experimental results when using loop reachability checks.

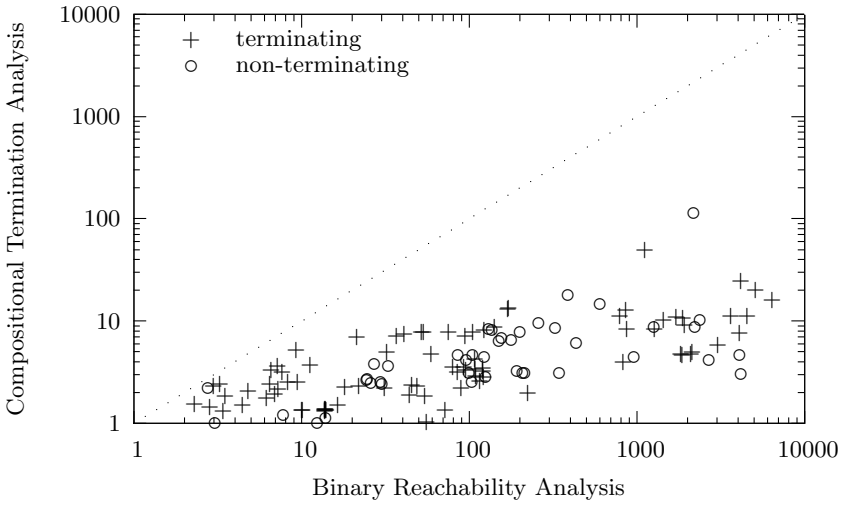

Fig. 5. Experimental results without loop reachability or precondition checks.

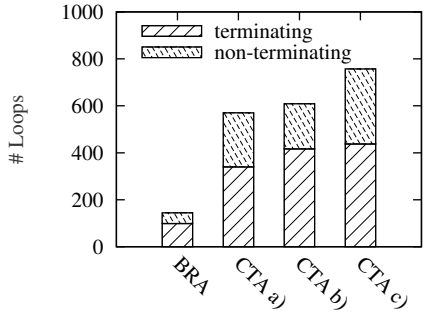

Fig. 6. Total number of loops analyzed within 2 hours per driver.

programming (e.g., [14]), term rewriting-based analysis (e.g., [15]), and functional programming (e.g., [16]).

We make use of a sub-procedure for ranking individual paths [4, 17, 6]; this problem is orthogonal to our contribution, which is focused on the iterative construction of a termination argument for a full program. We elaborate on the differences of our new algorithm and BRA as described in [3, 18, 1, 11].

The basis for reasoning about transition invariants, including the result that d.wf. transition invariants can be used to show termination, has been presented in [3]. The BRA algorithm was presented in [1]. We also make use of the results of [3], but develop them in a different direction: we show how to prove termination using the compositionality of transition invariants. Our algorithm passes smaller, loop-free fragments of the program to the safety checker, which enables it to outperform Binary Reachability Analysis.

In [11], the authors under-approximate the weakest precondition of paths to find a condition for termination. This result can be exploited in the context of our algorithm as well, as it allows for a generalization of path preconditions.

Berdine et al. present an algorithm for proving termination that is based on abstract interpretation [19]. Using an invariance analysis they construct a variance analysis, and they use the fact that the transitive closure of a well-founded relation is also well-founded to show that the fixed-point obtained by their analysis is correct. Their result may be used to improve the overall performance of our algorithm, as it can be modified to generate d.wf. transition invariants via abstraction.

Biere, Artho, and Schuppan propose an encoding of liveness properties into an assertion [20]. This approach allows proving termination of programs without a ranking sub-procedure. It has been reported to prove termination of programs that require non-linear ranking functions. Prior experimental results on our benchmarks indicate this encoding results in difficult safety checks [2].

7 Conclusion

The safety check is known as the bottleneck of Binary Reachability Analysis (BRA). We present a new algorithm for proving program termination that avoids this expensive safety check and is therefore able to outperform BRA. The latter relies on a safety checker to detect correctness of a disjunctively well-founded termination argument. We propose to check for compositionality of a candidate termination argument, which is much less expensive. To perform this test, our algorithm passes only loop-free segments of the program to a symbolic execution engine and, consequently, achieves much higher performance than other termination provers. In case the termination argument has to be refined, BRA uses a full counterexample path computed by a safety checker. In contrast, we pass an incrementally deeper unwinding of the loop to the rank finding procedure. Experimental results indicate an average speedup factor of 52 in comparison to BRA.

References

1. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, ACM (2006) 415–426
2. Cook, B., Kroening, D., Ruemmer, P., Wintersteiger, C.: Ranking function synthesis for bit-vector relations. In: TACAS, Springer (2010) 236–250
3. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, IEEE Computer Society (2004) 32–41
4. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: TACAS, Springer (2001) 67–81
5. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI, Springer (2004) 465–486
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: CAV, Springer (2005) 491–504
7. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: PADL, Springer (2007) 245–259
8. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, ACM (1988) 35–46
9. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: ATVA, Springer (2008)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, Springer (2004) 168–176
11. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: CAV. Volume 5123 of LNCS., Springer (2008) 328–340
12. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. **2** (1936) 230–265
13. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge (1949) 67–69
14. Codish, M., Taboch, C.: A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In: Alg. and Log. Programming, Springer (1997) 31–45
15. Thiemann, R., Giesl, J.: The size-change principle and dependency pairs for termination of term rewriting. Appl. Alg. in Eng., Comm. & Comp. **16** (2005) 229–270
16. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, ACM (2001) 81–92
17. Colón, M., Sipma, H.: Practical methods for proving program termination. In: CAV, Springer (2002) 442–454
18. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: SAS. Volume 3672 of LNCS., Springer (2005) 87–101
19. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. SIGPLAN Not. **42** (2007) 211–224
20. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66** (2002)