Automated nontermination proofs by safety proofs

Hong-Yi Chen[1], Byron Cook[1,2], Carsten Fuhs[1], Kaustubh Nimkar[1], and Peter O'Hearn[1]

1 **University College London**
Gower Street, London, United Kingdom
hongyichen00@gmail.com, c.fuhs@cs.ucl.ac.uk, k.nimkar@cs.ucl.ac.uk, p.ohearn@ucl.ac.uk

2 **Microsoft Research Cambridge**
21 Station Road, Cambridge, United Kingdom
bycook@microsoft.com

Abstract

We show how the problem of nontermination proving can be reduced to a question of under-approximation search guided by a safety prover. This reduction leads to new nontermination proving implementation strategies based on existing tools for safety proving. Our preliminary implementation has shown favorable results over existing tools.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, I.2.2 Automatic Programming

Keywords and phrases Nontermination analysis, Safety analysis, Closed recurrence set

1 Introduction

The problem of proving program *non*termination represents an interesting complement to termination as, unlike safety, termination's falsification cannot be witnessed by a finite trace. While the problem of proving termination has now been extensively studied, the search for reliable and scalable methods for proving nontermination remains open.

In this extended abstract we present a new method of proving nontermination based on a reduction to safety proving that leverages the power of existing tools. An iterative algorithm is developed which uses counterexamples to a fixed safety property to refine an underapproximation of a program. With our approach, existing safety provers can now be employed to prove nontermination of programs that previous techniques could not handle. Not only does the new approach perform better, it also leads to nontermination proving tools supporting features previous tools could not handle reliably, *e.g.* heap, nonlinear commands, and nondeterminism.

Recall that *safety* of a program means that no undesired (or *unsafe*) program state can be reached from any initial state of the program. On source code level, unsafe states can be expressed by a statement "`assert`(φ)" for a Boolean expression φ. Then the program is unsafe *iff* there exists a run of the program from an initial state such that in this run, φ is violated at the position of this statement. Techniques for safety proving include counter-example based abstraction refinement as in SLAM [1] or interpolation as in IMPACT [9].

Gupta *et al.* [8] characterize nontermination of a program by the existence of a *recurrence set*. A program is nonterminating *iff* there exists a recurrence set for the program's transition relation. The existence of a recurrence set implies that the program does not terminate when from a reachable state in the recurrence set we can always choose the next transition to a state that also belongs to the recurrence set.

As opposed to their approach we search for an underapproximation of the original program that *never* terminates, regardless of the values introduced by nondeterministic assignments during the run. This property is characterized by a closed recurrence set for the transition relation of the underapproximation. For every state in the closed recurrence set, every possible transition leads us to a state that belongs to the closed recurrence set. As "never terminates" can be encoded as safety property, we can then iterate a safety prover together with a method of underapproximating based on counterexamples. We have to be careful, however, to find the right underapproximation in order to avoid unsoundness.

We describe our algorithm informally. It takes as input a program P and a loop L in P to be considered for nontermination. We then mark the L's exit location as an error location and invoke a safety checker. Any path that reaches the exit location is the counterexample to safety and it cannot contribute towards the nontermination of the loop. We then find an underapproximation of P that eliminates this path. Our algorithm either finds a precondition for P or a precondition after a nondeterministic assignment statement such that every state which fulfills this precondition reaches the error location when the counterexample path is followed. To eliminate the counterexample path we then negate this condition and add a restriction on the state space to get our underapproximating refinement. We continue this procedure as long as there is some counterexample to safety of our current underapproximation.

Note that sometimes our refinements are too weak and the search for a safe underapproximation may lead to divergence. In such cases we use suitable heuristics to strengthen our underapproximation which then avoids the problem of divergence.

Let P' be our final underapproximation that is safe. We refer to the loop L after refinements as L'. To prove nontermination we first need to ensure that the loop L' in P' is still reachable after the refinements. This can again be encoded as a safety problem, this time marking the loop header as an error location. If safety is violated, the counterexample path represents the path to L' ensuring the reachability of L'.

Note that our refinements also restrict the choices for nondeterministic assignment statements. We finally ensure that for every reachable state at the nondeterministic assignment inside L', we can still make a choice so that execution is never halted. When the check succeeds, we report nontermination. In the final underapproximation, the set of all reachable states at the loop header of L' forms a closed recurrence set for the loop's transition relation.

2 Example

We now describe our algorithm using a simple example. Consider the following program. In this program the command $\mathsf{i} := \texttt{nondet}()$ represents nondeterministic value introduction into the variable i (*e.g.* user input). The loop in this program is nonterminating when the program is invoked with appropriate inputs and when appropriate choices for the `nondet` assignment are made. We are interested in automatically detecting this nontermination.

```
if (k ≥ 0)
    skip;
else
    i := −1;
while (i ≥ 0) {
    i := nondet();
}
i := 2;
```

In order to find the desired underapproximation for our example, we first introduce an `assume` statement (where "`assume`(φ)" can be implemented by "`if` ($\neg\varphi$) `exit`") at the beginning with the initial precondition `true`. We also place `assume(true)` statements after each use of `nondet`. We then put an `assert(false)` statement at points where the loop under consideration exits (thus encoding the "never terminates" property). See Figure 1(a).

We then use a safety checker (here: for programs on integer data) to search for paths that violate this assertion. Any error path clearly cannot contribute towards the nontermination

```
assume(true);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(a)

```
assume(k ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(b)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(true);
}

assert(false);

i := 2;
```
(c)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}

assert(false);

i := 2;
```
(d)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

assert(false);

while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}
```
(e)

```
assume(k ≥ 0 ∧ i ≥ 0);
assume(k ≥ 0);

skip;
while (i ≥ 0) {
   i := nondet();
   assume(i ≥ 0);
}
```
(f)

Figure 1 Original instrumented program (a) and its successive underapproximations (b), (c), (d). Reachability check for the loop (e), and nondeterminism-`assume` that must be checked for satisfiability (f).

of the loop. Initially, as a first counterexample to safety, we might get the path $\mathsf{k} < 0,\ \mathsf{i} := -1,\ \mathsf{i} < 0$, from a safety prover. We now want to determine from which states we can reach `assert(false)` and eliminate those states. Using a precondition computation similar to Calcagno *et al.* [4] we find the condition $\mathsf{k} < 0$. Note that our condition gives a set of states that actually reach the error location. To rule out the states $\mathsf{k} < 0$ we can add the negation (*e.g.* $\mathsf{k} \geq 0$) to the precondition `assume` statement. See Figure 1(b).

We then try to prove the `assert` statement unreachable for the program in Figure 1(b). Here we might get the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} < 0$, which again violates the assertion. For this path we would discover the precondition $\mathsf{k} \geq 0 \wedge \mathsf{i} < 0$, and to rule out these states we refine the precondition `assume` statement with "$\texttt{assume}(\mathsf{k} \geq 0 \wedge \mathsf{i} \geq 0)$". See Figure 1(c).

On this program our safety prover will again fail, perhaps resulting in the path $\mathsf{k} \geq 0,\ \texttt{skip},\ \mathsf{i} \geq 0,\ \mathsf{i} := \texttt{nondet}(),\ \mathsf{i} < 0$. In this case our algorithm stops computing the precondition at the command $\mathsf{i} := \texttt{nondet}()$. Here we would learn that at the nondeterministic command the result must be $\mathsf{i} < 0$ in order to violate the assertion, thus we would refine the assume statement just after the nondeterministic statement with the negation of $\mathsf{i} < 0$ and get "$\texttt{assume}(\mathsf{i} \geq 0)$". See Figure 1(d).

The program in Figure 1(d) cannot violate the assertion, and thus we have hopefully computed the desired underapproximation to the transition relation needed in order to prove nontermination. However, for soundness, it is essential to ensure that the loop in Figure 1(d) is still reachable, even after the successive restrictions to the state space. We encode this condition as a safety problem. See Figure 1(e). This time we add `assert(false)` before the loop and aim to prove that the assertion is violated. The existence of a path violating the assertion ensures that the loop in Figure 1(d) is reachable. In this case the assertion is reachable, and thus the loop is still reachable. The path violating the assertion is our desired path to the loop which we refer to as *stem*. Figure 1(f) shows the stem and the loop.

Finally we need to ensure that the `assume` statement in Figure 1(f) can always be satisfied with some choice of i by any reachable state from the restricted pre-state. This is necessary since our underapproximations may accidentally have eliminated not only the paths to the loop's exit location, but also all of the non-terminating paths inside the loop. We ensure this by calculating a location invariant *inv* before the `nondet` statement. We then check that the formula $\mathit{inv} \rightarrow \exists \mathsf{i}'.\mathsf{i}' \geq 0$ is valid. Even the weakest invariant `true` can be sufficient to easily prove the validity of the above formula. This ensures that for every reachable state at the nondeterministic assignment we can still make a choice so that execution is never halted. Once this check succeeds we report nontermination.

3 Experiments

In order to assess the impact of our approach, we have built a preliminary implementation within the tool T2 [2] [5] and evaluated it empirically comparing with the following tools:

- TNT [8]. Note that the original TNT tool was not available and thus we have reimplemented the underlying constraint-based algorithm with Z3 [6] as SMT backend.
- AProVE [7], using the Java Bytecode frontend. When proving nontermination of Java Bytecode programs, AProVE implements the SMT-based nontermination analysis by Brockschmidt *et al.* [3].
- Julia [13]: Julia implements an approach via a reduction from Java Bytecode to constraint logic programming described by Payet and Spoto [11].

As a benchmark set, we applied the tools on a set of 495 benchmarks from a variety of applications (*e.g.* Windows device drivers, the Apache web server, the PostgreSQL server,

integer approximations of numerical programs from a book on numerical recipes [12], integer approximations of benchmarks from LLBMC [10] and other tool evaluations).

We conducted three sets of experiments. The first set consists of all the 77 examples previously known to be nonterminating, the second set consists of all the 258 examples previously known to be terminating, and the third set consists of all the 160 examples for which no previous results are known and which are too large to render a manual analysis feasible. We used the first set of examples to assess the efficiency of the algorithm, the second set to demonstrate the algorithm's soundness, and the third set to check if our algorithm scales well on relatively large and complicated examples. The results demonstrate that our procedure is overwhelmingly the most successful tool and does not show erroneous behavior in our experiments.

4 Conclusion

In this paper we introduced a new method of proving nontermination. The idea is to split the reasoning in two parts: a safety prover is used to prove that a loop in an underapproximation of the original program *never* terminates; meanwhile failed safety proofs are used to calculate the underapproximation. Our implementation has shown that our approach leads to performance improvements against previous tools where they are applicable.

References

1 Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*, volume 2102 of *LNCS*, pages 260–264, 2001.

2 Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*, volume 8044 of *LNCS*, pages 413–429, 2013.

3 Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerException`s for Java Bytecode. In *Proc. FoVeOOS '11*, volume 7421 of *LNCS*, pages 123–141, 2012.

4 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26, 2011.

5 Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*, volume 7795 of *LNCS*, pages 47–61, 2013.

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, volume 4963 of *LNCS*, pages 337–340, 2008.

7 Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.

8 Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*, pages 147–158, 2008.

9 Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*, volume 4144 of *LNCS*, pages 123–136, 2006.

10 Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE '12*, LNCS, pages 146–161, 2012.

11 Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*, volume 253 of *ENTCS*, pages 83–96, 2009.

12 William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1989.

13 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.