Birkbeck UNIVERSITY OF LONDON

BIROn - Birkbeck Institutional Research Online

Chen, H.-Y. and Cook, B. and Fuhs, Carsten and Nimkar, K. and O'Hearn, P. (2014) Proving Nontermination via safety. In: Abraham, E. and Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes In Computer Science 8413. New York, U.S.: Springer, pp. 156-171. ISBN 9783642548611.

Downloaded from: https://eprints.bbk.ac.uk/id/eprint/13545/

Usage Guidelines:
Please refer to usage guidelines at https://eprints.bbk.ac.uk/policies.html or alternatively contact lib-eprints@bbk.ac.uk.

Proving nontermination via safety

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

[1] University College London
[2] Microsoft Research

Abstract. We show how the problem of nontermination proving can be reduced to a question of underapproximation 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 beats existing tools. Furthermore, our approach leads to easy support for programs with unbounded nondeterminism.

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 paper we develop 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 programs with nondeterminism, for which previous tools had only little support.

Limitations. Our proposed nontermination procedure can only prove nontermination. On terminating programs the procedure is likely to diverge (although some heuristics are proposed which aim to avoid this). While our method could be extended to further programming language features (*e.g.* heap, recursion), in practice the supported features of an underlying safety prover determine applicability. Our implementation uses a safety prover for non-recursive programs with linear integer arithmetic commands.

```
if (k ≥ 0)
  skip;
else
  i := −1;

while (i ≥ 0) {
  i := nondet();
}

i := 2;
```

Example. Before discussing our procedure in a formal setting, we begin with a simple example given to the right. In this program the command i := nondet() represents nondeterministic value introduction into the variable i. The loop in this program

is nonterminating when the program is invoked with appropriate inputs and when appropriate choices for `nondet` assignment are made. We are interested in automatically detecting this nontermination. The basis of our procedure is the search for an underapproximation of the original program that *never* terminates. As "never terminates" can be encoded as safety property (defined later as *closed recurrence* in Sect. 2), 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.

In order to find the desired underapproximation for our example, we introduce an `assume` statement 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 Fig. 1(a).

We can now use a safety checker to search for paths that violate this assertion. Any error path clearly cannot contribute towards the nontermination of the loop. After detecting such a path we calculate restrictions on the introduced `assume` statements such that the path is no longer feasible when the restriction is applied.

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.* [6] we find the condition $\mathsf{k} < 0$. The trick is to use the standard weakest precondition rule for assignments, but to use $\mathit{pre}(\texttt{assume}(Q), P) \triangleq P \wedge Q$ instead of the standard $\mathit{wp}(\texttt{assume}(Q), P) \triangleq Q \Rightarrow P$. This way, we only consider executions 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 Fig. 1(b).

In our procedure we try again to prove the assertion statement unreachable, using the program in Fig. 1(b). In this instance we might get the path $\mathsf{k} \geq 0$, `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 "`assume`($\mathsf{k} \geq 0 \wedge \mathsf{i} \geq 0$);". See Fig. 1(c).

On this program our safety prover will again fail, perhaps resulting in the path $\mathsf{k} \geq 0$, `skip`, $\mathsf{i} \geq 0$, $\mathsf{i} :=$ `nondet`(), $\mathsf{i} < 0$. Then our procedure would stop computing the precondition at the command $\mathsf{i} :=$ `nondet`() (for reasons discussed later). Here we would learn that at the nondeterministic command the result must be $\mathsf{i} < 0$ to violate the assertion, thus we would refine the `assume` statement just after the `nondet` with the negation of $\mathsf{i} < 0$: "`assume`($\mathsf{i} \geq 0$);" See Fig. 1(d).

The program in Fig. 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 Fig. 1(d) is still reachable, even after the successive restrictions to the state-space. We encode this condition as a safety problem. See Fig. 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 Fig. 1(d) is reachable. Here the assertion and thus the

(a)

```
assume(true);

if (k ≥ 0)
   skip;
else
   i := −1;

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

assert(false);

i := 2;
```

(b)

```
assume(k ≥ 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(true);
}

assert(false);

i := 2;
```

(d)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

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

assert(false);

i := 2;
```

(e)

```
assume(k ≥ 0 ∧ i ≥ 0);

if (k ≥ 0)
   skip;
else
   i := −1;

assert(false);

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

(f)

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

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

Fig. 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)**.

loop are still reachable. The path violating the assertion is our desired path to the loop which we refer to as *stem*. Fig. 1(f) shows the stem and the loop.

Finally we need to ensure that the `assume` statement in Fig. 1(f) can always be satisfied with some i by any reachable state from the restricted pre-state. This is necessary: our underapproximations may accidentally have eliminated not only the paths to the loop's exit location, but also all of the nonterminating paths inside the loop. Once this check succeeds we have proved nontermination.

2 Closed recurrence sets

In this section we define a new concept which is at the heart of our procedure, called *closed recurrence*. Closed recurrence extends the known concept of (open) recurrence [16] in a way that facilitates automation, e.g. via a safety prover.

Preliminaries. Let S be the set of states. Given a transition relation $R \subseteq S \times S$, for a state s with $R(s, s')$, we say that s' is a successor of s under R.

We will be considering programs P with finitely many program locations $\mathbb{L}$ and a set of memory states M, so the program's state space S is given as $S = \mathbb{L} \times M$. For instance, for a program on n integer variables, we have $M = \mathbb{Z}^n$, and a memory state amounts to a valuation of the program variables.

A program P on locations $\mathbb{L}$ is represented via its *control-flow graph (CFG)* $(\mathbb{L}, \Delta, l_i, l_f, l_e)$. The program locations are the CFG's nodes and Δ is a set of edges between locations labeled with commands. We designate special locations in $\mathbb{L}$: l_i is the initial location, l_f the final location, and l_e the error location. Each $(l, T, l') \in \Delta$ is a directed edge from l to l' labeled with a command T. We write $R_T \subseteq M \times M$ for the relation on memory states induced by T in the usual way.

We say that a memory state s at node l has a successor s' along the edge (l, T, l') *iff* $R_T(s, s')$ holds. A path π in a CFG is a sequence of edges (l_0, T_0, l_1) $(l_1, T_1, l_2) \ldots (l_{n-1}, T_{n-1}, l_n)$. The composite transition relation R_π of a path π is the composition $R_{T_0} \circ R_{T_1} \circ \ldots \circ R_{T_{n-1}}$ of the individual relations. We also describe a path π by the sequence of nodes it visits, e.g. $l_0 \to l_1 \to \ldots \to l_{n-1} \to l_n$.

Commands. We represent by V the set of all program variables. A deterministic assignment statement is of the form $\mathsf{i} := \mathsf{exp}$ where $\mathsf{i} \in V$ and exp is an expression over program variables. A nondeterministic assignment statement is of the form $\mathsf{i} := \mathtt{nondet}(); \mathtt{assume}(\mathsf{Q});$ where $\mathsf{i} \in V$, $\mathtt{nondet}()$ is a nondeterministic choice and Q is a boolean expression over V representing the restriction that the $\mathtt{nondet}()$ choice must obey. Conditional statements are encoded using `assume` commands (from Nelson [22]): $\mathtt{assume}(\mathsf{Q})$, where Q is a boolean expression over V. W.l.o.g. l_i has 0 incoming and 1 outgoing edge, labeled with $\mathtt{assume}(\mathsf{Q})$, where initially usually $\mathsf{Q} \triangleq \mathtt{true}$. For readability, in our example CFGs we often write Q for $\mathtt{assume}(\mathsf{Q})$. Our algorithm will later strengthen Q in the `assume`-statements.

We define the indegree of a node l in a CFG to be the number of incoming edges to l. Similarly, the outdegree of a node l in a CFG is the number of outgoing edges from l. A node $l \in \mathbb{L} \setminus \{l_i, l_f, l_e\}$ must be of one of the following types.

1. A deterministic assignment node: l has outdegree exactly 1 and the outgoing edge is labeled with a deterministic assignment statement or `skip`. Any memory state s at l has a unique successor s' along the edge.
2. A deterministic conditional node: l has outdegree 2 with one edge labeled $\mathtt{assume}(\varphi)$, the other edge labeled $\mathtt{assume}(\neg\varphi)$. Any memory state s at l has a unique successor s' along one edge and no successor along the other edge.
3. A nondeterministic assignment node: l has outdegree exactly 1 and the outgoing edge is labeled with a nondeterministic assignment statement. A memory state s at l may have zero or more successors along the outgoing edge depending on the condition present in the $\mathtt{assume}(\mathsf{Q})$ statement.

In our construction of a CFG, a lone statement assume(Q) from an input program is modeled by a deterministic conditional node, with one edge labeled assume(Q) and the other edge labeled assume(¬Q) leading to the end location l_f.

Example. Consider the CFG for our initial example given in Fig. 2. Here we have the initial location $l_i = 0$, the final location $l_f = 7$, and the error location $l_e = 8$. The nodes 2, 3 and 6 are deterministic assignment nodes, nodes 1 and 4 are deterministic conditional nodes, and node 5 is a nondeterministic assignment node.

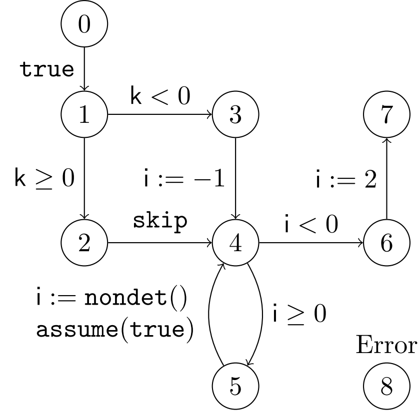


Fig. 2. CFG for initial example

CFG loops. Given a program with its CFG, a loop L in the CFG is a set of nodes s.t.

- There exists a path from any node of L to any other node of L.
- (W.l.o.g.) we assume that there is only one node h of L s.t. there exists a node $n \notin L$ with an edge from n to h. The node h is called the *header node* of L.

The subgraph of CFG containing all nodes of L is called the *loop body* of L. Header h of L is a deterministic conditional node with one edge that is part of the loop body, the *guard edge* of L. The other edge of h goes to a node $e \notin L$. We call this edge the *exit edge* of L and e the *exit location* of L.[3]

Example. In Fig. 2, the only loop is $L = \{4, 5\}$, and its header node h is 4. The exit location of L is 6.

Given a loop L in program P, we define a *loop path* π_L as any finite path through L's body of the form $(l_0 = l_h) \to l_1 \to ... \to l_{n-1} \to (l_n = l_h)$, where l_h is the header node of L and $\forall p.(0 < p < n) \to l_p \neq l_h$. We define the *composite transition relation* R_L of L as $R_L(s, s')$ *iff* there exists a loop path π s.t. $R_\pi(s, s')$. Here R_L can be an infinite (or empty) set. The initial states I_L for R_L are the set of reachable states at the loop header before the loop is entered for the first time.

Preconditions. When computing preconditions of assume statements we borrow from Calcagno *et al.* [6]: $pre(\texttt{assume}(Q), P) \triangleq P \wedge Q$, called the "assume as assert trick". This lets us interpret assume statements (often from conditional branches) in a way that allows us to determine in a precondition the states from which an error location can be reached in a safety counterexample path. For assignment statements we will use the standard weakest precondition [12].

Example. Note that the weakest precondition of an assignment with nondeterminism is a little subtle. Let i := nondet(); assume(true); be the nondet statement under consideration. The weakest precondition for the postcondition $(\mathsf{i} < \mathsf{j})$ is false (equivalent to $\forall \mathsf{i}.\ \mathsf{i} < \mathsf{j}$). However the weakest precondition for the postcondition $(\mathsf{i} < \mathsf{j} \vee \mathsf{k} > 0)$ is $(\mathsf{k} > 0)$.

[3] Languages like Java or C allow loops with additional exit edges also from locations other than the loop header, which are implemented by commands like break or goto. We also support such more general loops, via a program transformation.

Recurrence sets. A relation R with initial states I is nonterminating *iff* there exists an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ with $s_0 \in I$. Gupta *et al.* [16] characterize nontermination of a relation R by the existence of a *recurrence set*, *viz.* a nonempty set of states $\mathcal{G}$ such that for each $s \in \mathcal{G}$ there exists a transition to some $s' \in \mathcal{G}$. In particular, an infinite transition sequence $s_0 \xrightarrow{R} s_1 \xrightarrow{R} s_2 \xrightarrow{R} \ldots$ itself gives rise to the recurrence set $\{s_0, s_1, s_2, \ldots\}$. Here we extend the notion of a recurrence set to include initial states. A transition relation R with initial states I has an (open) *recurrence set* of states $\mathcal{G}(s)$ *iff* (1) and (2) hold. A transition relation R with initial states I is nonterminating *iff* it has a recurrence set of states.

$$\exists s. \mathcal{G}(s) \wedge I(s) \quad (1)$$

$$\forall s \exists s'. \mathcal{G}(s) \rightarrow R(s, s') \wedge \mathcal{G}(s') \quad (2)$$

A set $\mathcal{G}$ is a *closed recurrence set* for a transition relation R with initial states I *iff* the conditions (3)–(5) hold. In contrast to open recurrence sets, we now require a purely universal property: for each $s \in \mathcal{G}$ and *for each* of its successors s', also s' must be in the recurrence set (Condition (5)). So instead of requiring that we *can* stay in the recurrence set, we now demand that we *must* stay in the recurrence set. This now helps us to incorporate nondeterministic transition systems too.

$$\exists s. \mathcal{G}(s) \wedge I(s) \quad (3)$$

$$\forall s \exists s'. \mathcal{G}(s) \rightarrow R(s, s') \quad (4)$$

$$\forall s \forall s'. \mathcal{G}(s) \wedge R(s, s') \rightarrow \mathcal{G}(s') \quad (5)$$

Now what if a state s in our recurrence set $\mathcal{G}$ has no successor s' at all? Our alleged infinite transition sequence would reach a sudden halt, yet our universal formula would trivially hold. Thus, we impose that each $s \in \mathcal{G}$ has *some* successor s' (Condition (4)). But this existential statement need not mention that s' must be in $\mathcal{G}$ again—our previous *universal* statement already takes care of this.

Theorem 1 (Closed Recurrence Sets are Recurrence Sets). *Let $\mathcal{G}$ be a closed recurrence set for R with initial states I. Then $\mathcal{G}$ is also an (open) recurrence set for R with initial states I.*

The proofs for all theorems can be found in the technical report [7]. □

Underapproximation. We call a transition relation R' with initial states I' an underapproximation of transition relation R with initial states I *iff* $R' \subseteq R$, $I' \subseteq I$. Then *every* nonterminating program contains a closed recurrence set as an underapproximation (*i.e.*, together with underapproximation, closed recurrence sets characterize nontermination).

Theorem 2 (Open Recurrence Sets Always Contain Closed Recurrence Sets). *There exists a recurrence set $\mathcal{G}$ for a transition relation R with initial states I* iff *there exist an underapproximation R' with initial states I' and $\mathcal{G}' \subseteq \mathcal{G}$ such that $\mathcal{G}'$ is a closed recurrence set for R' with initial states I'.*

3 Algorithm

Our nontermination proving procedure Prover is detailed in Fig. 3. Its input is a program P given by its CFG, and a loop to be considered for nontermination. To prove nontermination of the entire program P we need to find only one nonterminating loop L. This can be done in parallel. Alternatively, the procedure

```
PROVER (CFG P, Loop L)
    h := header node of L
    e := exit node of L in P
    P' := UNDERAPPROXIMATE(P, e)
    L' := refined loop L in P'
    if ¬ REACHABLE(P', h) then
        return Unknown, ⊥
    fi
    Π := {π | π feasible path to h in P'}
    for all π ∈ Π do
        P' := π :: L' // concatenation
        if VALIDATE(P') then
            return NonTerminating, P'
        fi
    done
    return Unknown, ⊥
```

```
UNDERAPPROXIMATE (CFG P, Node e)
    κ := [ ]
    while REACHABLE(P, e) do
        π := feasible path to e in P
        κ := π :: κ
        P := REFINE(P, π)
        if the n most recent paths in κ
            are repeating then
            P := STRENGTHEN(P, FIRST(κ))
        fi
    done
    return P
```

```
REFINE (CFG P, Path π)
    (l_0 T_0 l_1)(l_1 T_1 l_2) ... (l_{n-1} T_{n-1} l_n) := π
    Calculate WPs ψ_1, ψ_2 ... ψ_{n-1} along π
        so {ψ_1}T_1{ψ_2}T_2 ... {ψ_{n-1}}T_{n-1}{true}
        are valid Hoare-triples.
    Find p s.t. ψ_p ≠ false ∧ ∀q < p. ψ_q = false
    P := P|_(T_{p-1}, ¬ψ_p)
    return P
```

```
STRENGTHEN (CFG P, Path π)
    (l_0 T_0 l_1)(l_1 T_1 l_2) ... (l_{n-1} T_{n-1} l_n) := π
    Calculate WPs ψ_1, ψ_2 ... ψ_{n-1} along π
        so {ψ_1}T_1{ψ_2}T_2 ... {ψ_{n-1}}T_{n-1}{true}
        are valid Hoare-triples.
    Find p s.t. ψ_p ≠ false ∧ ∀q < p. ψ_q = false
    W := {v | v gets updated in subpath
        (l_p T_p l_{p+1}) ... (l_{n-1} T_{n-1} l_n)}
    ρ_p := QE(∃W. ψ_p)
    P := P|_(T_{p-1}, ¬ρ_p)
    return P
```

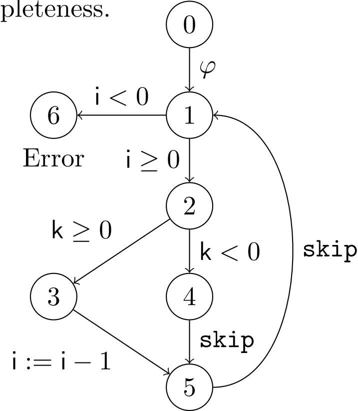
```
VALIDATE (CFG P')
    L' := the outermost loop in P'
    M := {l | l is nondet assignment node in L'}
    for all l ∈ M do
        Calculate invariant inv_l at node l in P'
        let nondet statement at l be
            v := nondet(); assume(φ);
        if inv_l → ∃v. φ is not valid then
            return false
        fi
    done
    return true
```

Fig. 3. Algorithm PROVER for underapproximation to synthesize a reachable nonterminating loop. To prove nontermination of P, PROVER should be run on all loops L.

can be implemented sequentially, but then timeouts are advisable in PROVER, as the procedure might diverge and cause another loop to not be considered.

The subprocedure UNDERAPPROXIMATE performs the search for an underapproximation such that we can prove the loop is never exited. While the loop exit is still REACHABLE (*a.k.a.* "unsafe"), we use the subprocedure REFINE to examine paths returned from an off-the-shelf safety prover. Here the notation $P|_{(T_i,\varphi)}$ denotes P with an additional `assume`(φ) added to the transition T_i. From the postcondition `true` (used to indicate success in reaching the loop exit), we use a backwards precondition analysis to find out which program states will inevitably end up in the loop exit. We continue this precondition calculation until either we have reached the beginning of the path or until just before we have reached a nondeterministic assignment that leads to the precondition `false`. We then negate this condition as our underapproximating refinement.

In some cases our refinement is too weak, leading to divergence. The difficulty is that in cases the same loop path will be considered repeatedly, but at each instance the loop will be unrolled for an additional iteration. To avoid this problem we impose a limit n for the number of paths that go along the same locations (possibly with more and more repetitions). We call such paths *repeating.* If we reach this limit, we use the subprocedure STRENGTHEN to strengthen the precondition, inspired by a heuristic by Cook and Koskinen [8]. Here we again calculate a precondition, but when we have found ψ_p, we quantify out all the variables that are written to after ψ_p and apply quantifier elimination (QE) to get ρ_p. We then refine with $\neg\rho_p$. This leads to a more aggressive pruning of the transition relation. This heuristic can lead to additional incompleteness.



Example. Consider the instrumented program to the right. Suppose we have initially $\varphi \triangleq \mathsf{i} \geq 0$. We might get $\text{cex}_1 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as a first counterexample. The REFINE procedure finds the weakest precondition $\mathsf{k} \geq 0 \wedge \mathsf{i} = 0$ at location 1. Adding its negation to φ and simplifying the formula gives us $\varphi \triangleq (\mathsf{i} \geq 0) \wedge (\mathsf{k} < 0 \;\vee\; \mathsf{i} \geq 1)$. Now we may get $\text{cex}_2 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as next counterexample, and REFINE updates $\varphi \triangleq (\mathsf{i} \geq 0) \wedge (\mathsf{k} < 0 \;\vee\; \mathsf{i} \geq 2)$. Now we may get $\text{cex}_3 : 0 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 2 \to 3 \to 5 \to 1 \to 6$ as next counterexample. Note that cex_1, cex_2, cex_3 are repeating counterexamples and if we just use the REFINE procedure, UNDERAPPROXIMATE gets stuck in a sequence of infinite counterexamples. Now STRENGTHEN identifies the repeating counterexamples, considers cex_1 and calculates the weakest precondition $\psi_1 \triangleq \mathsf{k} \geq 0 \wedge \mathsf{i} = 0$. It then existentially quantifies out variable i as it gets modified later along cex_1. We get $\exists \mathsf{i}.\ \mathsf{k} \geq 0 \wedge \mathsf{i} = 0$, and quantifier elimination yields $\rho_1 \triangleq \mathsf{k} \geq 0$. Clearly ψ_1 entails ρ_1. Adding $\neg\rho_1$ to φ and simplifying the formula we get $\varphi \triangleq \mathsf{i} \geq 0 \wedge \mathsf{k} < 0$. Now all repeating counterexamples are eliminated, the program is safe, and we have obtained a closed recurrence set witnessing nontermination of the original program.

In the UNDERAPPROXIMATE procedure, once there are no further counterexamples to safety of P, we know that in P the loop exit is not reachable. The procedure returns the final underapproximation (denoted by P') that is safe.

When UNDERAPPROXIMATE returns to PROVER, we check if in P' the original loop L after refinements has a closed recurrence set. We refer to the refined loop as L'. In order to check the existence of a closed recurrence set, we first need to ensure that L' is reachable in P' even after the refinements. We again pose this problem as a safety/reachability problem. This time we mark the header node of L' as an error location in P' and hope that P' is unsafe. If P' is safe then clearly we have failed to prove nontermination and we report the result as unknown. If P' is unsafe, then the counterexample to its safety is a path to the header of L'. We enumerate all such paths to the header of L' in a set Π (generated lazily

in our implementation). For each such path $\pi \in \Pi$ we then create a simplified CFG P' by concatenating π to L', thus eliminating other paths to L'.

At this point, we are sure that the header of L' is reachable and there is no path that can reach the exit location of L'. However refinements in UNDERAPPROXIMATE may have restricted the `nondet` statements inside L' by strengthening the `assume` statements associated with them. Thus a reachable state at the nondeterministic assignment node may not have a successor along its outgoing edge. This would bring our alleged infinite execution to a halt. The safety checker cannot detect this since then the path just gets blocked at this node, and the error location at the exit of L' cannot be reached.

Example. Consider the instrumented program in Fig. 4. Suppose initially $\varphi \triangleq \texttt{true}$. The original program (without instrumentation) is clearly terminating. Our algorithm might give $cex_1 : 0 \to 1 \to 2 \to 3 \to 4 \to 6 \to 1 \to 7$ as the first counterexample. The `nondet` statement at node 2 gets restricted by updating $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9)$. Now we might get $cex_2 : 0 \to 1 \to 2 \to 3 \to 5 \to 6 \to 1 \to 7$ as the next counterexample. Our algorithm restricts the `nondet` statement at node 2 by updating $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} == 11)$.

However now there are no further counterexamples, and the safety checker returns safe. The state $s \models \mathsf{i} = 10$ at node 2 has no successor along the outgoing edge as there is no way to satisfy the condition φ and the execution is halted, so it would be unsound to report the result as Nonterminating.

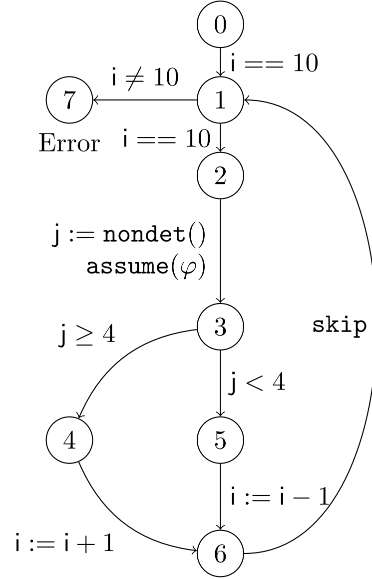


Fig. 4. Program showing why we need VALIDATE procedure

Note that at first it may appear that adding another outgoing edge to node 2 with $\mathsf{j} := \texttt{nondet}(); \texttt{assume}(\neg\varphi)$; and marking the next node as an error node would help us catch the halted execution. However the problem is that this would discover again all of the previously eliminated counterexamples as well. Thus we need a special check by the VALIDATE procedure, which we describe next.

VALIDATE takes as input the final underapproximation P'. It first calculates a location invariant at every nondet. assignment node inside the outermost loop L' in P'. Let l be a nondet. assignment node with: $\mathsf{v} := \texttt{nondet}(); \texttt{assume}(\varphi)$; Let inv be a location invariant at l. VALIDATE then checks if (6) is valid. This formula checks if for all reachable states at l, a choice can be made for the `nondet` assignment obeying φ (and thus Condition (4) holds). VALIDATE returns `true` *iff* (6) holds for all `nondet` statements in L'.

$$inv \to \exists \mathsf{v}.\, \varphi \quad (6)$$

Example. Consider the program in Fig. 1(f). Using a standard invariant generator we calculate the invariant $\mathsf{i} \geq 0$ before line 4. Substituting in (6) we get, $\mathsf{i} \geq 0 \to \exists \mathsf{i}'.\, \mathsf{i}' \geq 0$. Clearly the formula is valid. Note that in most of the cases

even the weakest invariant `true` can be sufficient to prove validity of (6). In this example as well we can easily prove that $\texttt{true} \rightarrow \exists \mathsf{i}'.\, \mathsf{i}' \geq 0$ is valid.

Moreover, consider the program in Fig. 4. Suppose $\varphi \triangleq (\mathsf{j} \leq 3 \vee \mathsf{i} == 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} == 11)$. Using an invariant generator, we obtain the location invariant $\mathsf{i} = 10$ at location 2. Then (6) becomes $\mathsf{i} = 10 \rightarrow \exists \mathsf{j}.\, (\mathsf{j} \leq 3 \vee \mathsf{i} = 9) \wedge (\mathsf{j} \geq 4 \vee \mathsf{i} = 11)$. Clearly the formula is not valid. In this case VALIDATE returns `false`.

If VALIDATE returns `true`, we are sure that every reachable state at the nondeterministic assignment node in L' has a successor along the edge. At this point, we report nontermination and return the final underapproximation P' of P as a proof of nontermination for P: P' is a closed recurrence set.

Note that as invariants are overapproximations, we may report unknown in some cases even when the discovered underapproximation actually does have a closed recurrence set. However, the check is essential to retain soundness.

Theorem 3 (Correctness of Prover for Nontermination). *Let P be a program and L a loop in P. Suppose* PROVER $(P, L) = \mathsf{Nonterminating}, P'$. *Then P is nonterminating.*

4 Nested Loops

```
if (i == 10) {
   while (i > 0) {
      i := i − 1;
      while (i == 0)
         skip;
   }
   assert(false);
}
```

Our algorithm can handle nested loops easily. That is a part of the beauty of the reduction to safety, as existing safety provers (*e.g.* SLAM, IMPACT, *etc.*) handle nested loops with ease. Note that technically we only need to consider an outermost loop. Consider the instrumented program with nested loops to the right. Here the outer loop decreases the value of i 10 times and then it is the inner loop that is nonterminating. However, it suffices only to consider the outermost loop for safety as the `assert(false)` at the end of the outer loop is not reachable, but the head of the outer loop is reachable, so that we have proved nontermination.

```
assume(φ);
while (k ≥ 0) {
   k := k + 1;
   j := k;
   while (j ≥ 1)
      j := j − 1;
}
assert(false);
```

Tricky example. We close with an example that partially explains the advantage seen for our tool over TNT (discussed in Sect. 5). Consider the program to the right (already shown with our algorithm's instrumentation, initially with $\varphi \triangleq \texttt{true}$). This program clearly does not terminate, yet TNT will fail to prove it. In fact, our implementation of TNT which follows the strategy discussed for enumerating lassos [16] diverges looking at larger and larger cyclic paths (*i.e.* straight-line code from a location back to that location). The difficulty here is that each cyclic path is well-founded. Consider *e.g.* this cyclic path from the head h of the outer loop back to h:

$$\mathsf{k} \geq 0,\ \ \mathsf{k} := \mathsf{k} + 1,\ \ \mathsf{j} := \mathsf{k},\ \ \mathsf{j} \geq 1,\ \ \mathsf{j} := \mathsf{j} - 1,\ \ \mathsf{j} < 1,\ \ \mathsf{k} \geq 0$$

This path is well-founded. In fact the path cannot be repeated. The root of the problem is the command sequence $\mathsf{j} \geq 1,\ \mathsf{j} := \mathsf{j} - 1,\ \mathsf{j} < 1$, which tells us that j goes from exactly 1 to 0. Because $\mathsf{j} = 1$ before entering the inner loop, we know

that $\mathsf{k} = 1$, thus by the $\mathsf{k} := \mathsf{k} + 1$ command we know that $\mathsf{k} = 0$ at the start of the loop's execution. Thus the command sequence respects the following condition: $\mathsf{k}' > \mathsf{k} \wedge \mathsf{k}' \leq 1$, and thus it is well-founded. Hence, the lasso-based tool TNT will never be able to prove nontermination of this program. The tool APROVE cannot prove such *aperiodic* nontermination for nested loops either [5].

Our approach, however, does not fall victim to this problem. It will find the path: $\mathsf{k} < 0$, resulting in $\varphi \triangleq \mathsf{k} \geq 0$. As `assert(false)` is unreachable with this restriction (and the loop is still reachable), we have proved nontermination.

5 Related work

Automatic tools for proving nontermination of term rewriting systems include [14, 23]. However, while nontermination analysis for term rewriting considers the *entire* state space as legitimate initial states for a (possibly infinite) evaluation sequence, our setting also factors in reachability from the initial states.

Static nontermination analysis has also been investigated for logic programs (*e.g.* [24, 31]). Most related to our setting are techniques for constraint-logic programs (CLPs) [25]. Termination tools for CLPs (*e.g.* [25]) can in cases be used to prove nontermination of imperative programs (*e.g.* JULIA [26] can show nontermination for Java Bytecode programs if the abstraction to CLPs is exact, but gives no witness like a recurrence set to the user). The main difficulty for imperative programs is that typically overapproximating abstractions (in general unsound for nontermination) are used for converting languages like Java and C to CLPs.

TNT [16] uses a characterization of nontermination by recurrence sets. We build upon this notion and introduce *closed* recurrence sets in our formalization, as an intermediate concept during our nontermination proof search. In contrast to us, TNT is restricted to programs with *periodic* "lasso-shaped" counterexamples to termination. We support unbounded nondeterminism in the program's transition relation, whereas TNT is restricted to deterministic commands.

The tool INVEL [30] analyzes nontermination of Java programs using a combination of theorem proving and invariant generation. However, INVEL does not provide a witness for nontermination. Like Brockschmidt *et al.* [5], we were unable to obtain a working version of INVEL. Note that in the empirical evaluation by Brockschmidt *et al.* [5], the APROVE tool (which we have compared against) subsumed INVEL on INVEL's data set. Finally, INVEL is only applicable to deterministic (integer) programs, yet our approach allows nondeterminism as well.

Atig *et al.* [1] describe a technique for proving nontermination of multithreaded programs, via a reduction to nontermination reasoning for sequential programs. Our work complements Atig *et al.*, as we provide improvements to the underlying sequential tools that future multithreaded tools can make use of.

The tool TREX [19] combines existing nontermination proving techniques with a TERMINATOR-like [9] iterative procedure. Our new method should complement TREX nicely, as ours is more powerful than the underlying nontermination proving approach previously used [16].

AProVE [13] uses SMT to prove nontermination of Java programs [5]. First nontermination of a loop regardless of its context is shown, then reachability of this loop with suitable values. Drawbacks are that they require recurrence sets to be singletons (after program slicing) or the loop conditions to be invariants.

Gurfinkel *et al.* [18] present the CEGAR-based model checker Yasm which supports arbitrary CTL properties, such as $\mathsf{EG}\ pc \neq \mathsf{END}$, denoting nontermination. Yasm implements a method of both under and over-approximating the input program. Unfortunately, together with the author of Yasm we were not able to get the tool working on our examples [17]. We suspect that our approach will be faster, as it uses current safety proving techniques, *i.e.* Impact [20] rather than Slam-style technology [2]. This is a feature of our approach: *any* off-the-shelf software model checker can be turned into a nontermination prover.

Nontermination proving for finite-state systems is essentially a question of safety [3]. Nontermination and/or related temporal logics are also supported for more expressive systems, *e.g.* pushdown automata [28].

Recent work on CTL proving for programs uses an off-the-shelf nontermination prover [8]. We use a few steps when treating nondeterminism which look similar to the approach from [8]. The key difference is that our work *provides* a nontermination prover, whereas the previous work *requires* one off-the-shelf.

Gulwani *et al.* [15] (Claim 3), make a false claim that is similar to our own. Their claim is false, as a nondeterministic program can be constructed which represents a counterexample. Much of the subtlety in our approach comes from our method of dealing with nondeterminism.

6 Experiments

We have built a preliminary implementation of our approach within the tool T2 [10, 4][4] and conducted an empirical evaluation with it against these tools:

- TNT [16], the original TNT tool was not available, and thus we have reimplemented its constraint-based algorithm with Z3 [11] as SMT backend.
- AProVE [13], via the Java Bytecode frontend, using the SMT-based nontermination analysis by Brockschmidt *et al.* [5].
- Julia [29], which implements an approach via a reduction to constraint logic programming described by Payet and Spoto [26].

As a benchmark set, we used a set of 492 benchmarks for termination analysis from a variety of applications also used in prior tool evaluations (*e.g.* Windows device drivers, the Apache web server, the PostgreSQL server, integer approximations of numerical programs from a book on numerical recipes [27], integer approximations of benchmarks from LLBMC [21] and other tool evaluations).[5]

Of these, 81 are known to be nonterminating and 254 terminating. For 157 examples, the termination status is unknown. These examples include a program whose termination would imply the Collatz conjecture, and the remaining examples are too large to render a manual analysis feasible. On average a CFG in

[4] We will make our implementation available in the next source code release of T2.
[5] Download: `http://www0.cs.ucl.ac.uk/staff/C.Fuhs/safety-nontermination`

	(a)			(b)			(c)		
	Nonterm	TO	No Res	Nonterm	TO	No Res	Nonterm	TO	No Res
Fig. 3	51	0	30	0	45	209	82	3	72
AProVE	0	61	20	0	142	112	0	139	18
Julia	3	8	70	0	40	214	0	91	66
TNT	19	3	59	0	48	206	32	12	113

Fig. 5. Evaluation success overview, showing the number of problems solved for each tool. Here **(a)** represents the results for known nonterminating examples, **(b)** is known terminating examples, **(c)** is (previously) unknown examples.

our test suite has 18.4 nodes (max. 427 nodes) and 2.4 loops (max. 120 loops).

Unfortunately each tool requires a different machine configuration, and thus a direct comparison is difficult. Experiments with our procedure were performed on a dualcore Intel Core 2 Duo U9400 (1.4 GHz, 2 GB RAM, Windows 7). TNT was run on Intel Core i5-2520M (2.5 GHz, 8 GB RAM, Ubuntu Linux 12.04). We ran AProVE on Intel Core i7-950 (3.07 GHz, 6 GB RAM, Debian Linux 7.2). Note that the TNT-/AProVE-machines are significantly faster than the machine our new procedure was run on, thus we can make some adjusted comparison between the tools. For Julia, an unknown cloud-based configuration was used. All tools were run with a timeout of 60s. When a tool returned early with no definite result, we display this in the plots using the special "NR" (no result) value.

We ran three sets of experiments: (a) all the examples previously known to be nonterminating, (b) all the examples previously known to be terminating, and (c) all the examples where no previous results are known. With (a) we assess the efficiency of the algorithm, (b) is used to demonstrate its soundness, and (c) checks if our algorithm scales well on relatively large and complicated examples. The results of the three sets of experiments are given in Fig. 5, which shows for each tool and for each set (a)–(c) the numbers of benchmarks with nontermination proofs ("Nonterm"), timeouts ("TO"), and no results ("No Res"). (Proofs of *termination*, found by AProVE and Julia, are also listed as "No Res".)

On the 89 deterministic instances of our benchmark set, our implementation proves nontermination of 33 examples, and TNT of 21 examples. We have also experimented with different values for the number of repeated paths before invoking Strengthen. The results are reported in Fig. 6 (runtimes are for successful nontermination proofs).

# Paths	Nonterm	Time [s]
2	133	272
4	133	301
6	129	264
∞	123	272

Fig. 6. Repeated paths before calling Strengthen

Fig. 7 charts the difference in power and performance between our implementation and TNT in a scatter plot, in log scale. Here we have included all programs from (a)–(c). Each '×'-mark in the plot represents an example from the benchmark. The value of the x-axis plots the runtime of TNT and the y-axis value plots the runtime of our procedure on the same example. Points under the diagonal are in favor of our procedure. Thus, the more '×'-marks there are in the lower-righthand corner, the better our tool has performed.

Discussion. Figs. 5(a&c) demonstrate that our technique is overwhelmingly the most successful tool (Fig. 5(b) confirms simply that no tool has demonstrable

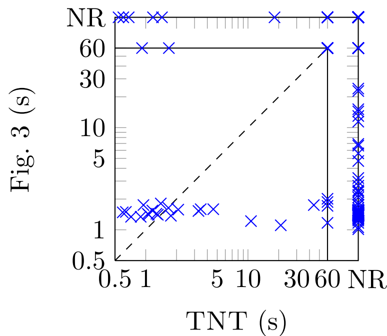


Fig. 7. Evaluation results of our procedure vs. TNT. Scatter plot in log scale. Timeout=60s. NR="No Result", indicating failure of the tool.

soundness bugs). The poor precision of APROVE & JULIA is mainly due to the nondeterministic updates originally present in many of the benchmarks and also introduced by the (automated) conversion of the benchmarks to Java (the two tools' input syntax). This shows the lack of reliable support of nondeterminism in today's nontermination tools.

The TNT algorithm requires outright that nondeterminism must not occur in the input. Our implementation of TNT softens this requirement slightly: parts of the program with `nondet`-assignments are allowed as long as they are not used during the synthesis of recurrence sets.

Finally, we observe in Fig. 6 that the STRENGTHEN procedure provides additional precision for our approach without harming performance.

7 Conclusion

We have 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. We have shown that nondeterminism can be easily handled in our framework while previous tools often fail. Furthermore, we have shown that our approach leads to performance improvements against previous tools where they are applicable.

Our technique is not restricted to linear integer arithmetic: Given suitable tools for safety proving and for precondition inference, in principle our approach is applicable to *any* program setting (note that the STRENGTHEN procedure is just an optimization). For future work, *e.g.* heap programs are a highly promising candidate for nontermination analysis via abduction tools for separation logic [6].

Acknowledgments. We thank Marc Brockschmidt, Fabian Emmes, Florian Frohn and Fausto Spoto for help with the experiments and Tony Hoare, Jules Villard and the anonymous reviewers for insightful comments.

References

1. Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. Detecting fair non-termination in multithreaded programs. In *Proc. CAV '12*.
2. Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Proc. CAV '01*.
3. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. In *Proc. FMICS '02*.
4. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.

5. Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerException`s for Java Bytecode. In *Proc. FoVeOOS '11*.
6. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
7. Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. Proving nontermination via safety. Technical Report RN/13/23, UCL, 2014.
8. Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *Proc. PLDI '13*.
9. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *Proc. CAV '06*.
10. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *Proc. TACAS '13*.
11. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*.
12. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
13. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*.
14. Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*.
15. Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proc. PLDI '08*.
16. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proc. POPL '08*.
17. Arie Gurfinkel. Private communication. 2012.
18. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Proc. CAV '06*.
19. William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In *Proc. SAS '10*.
20. Ken McMillan. Lazy abstraction with interpolants. In *Proc. CAV '06*.
21. Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE '12*.
22. Greg Nelson. A generalization of Dijkstra's calculus. *ACM TOPLAS*, 11(4), 1989.
23. Étienne Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theor. Comput. Sci.*, 403(2-3), 2008.
24. Étienne Payet and Frédéric Mesnard. Nontermination inference of logic programs. *ACM TOPLAS*, 28(2), 2006.
25. Étienne Payet and Frédéric Mesnard. A non-termination criterion for binary constraint logic programs. *TPLP*, 9(2), 2009.
26. Étienne Payet and Fausto Spoto. Experiments with non-termination analysis for Java Bytecode. In *Proc. BYTECODE '09*.
27. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge Univ. Press, 1989.
28. Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In *Proc. TACAS '12*.
29. Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.
30. Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Proc. TAP '08*.
31. Dean Voets and Danny De Schreye. A new approach to non-termination analysis of logic programs. In *Proc. ICLP '09*.