Synthesizing Ranking Functions from Bits and Pieces*

Caterina Urban[1,2], Arie Gurfinkel[2], and Temesghen Kahsai[2,3]

[1] ETH Zürich, Switzerland
[2] Carnegie Mellon University, USA
[3] NASA Ames Research Center, USA

Abstract. In this work, we present a novel approach based on recent advances in software model checking to synthesize ranking functions and prove termination (and non-termination) of imperative programs.
Our approach incrementally refines a termination argument from an under-approximation of the terminating program state. Specifically, we learn *bits* of information from terminating executions, and from these we extrapolate ranking functions over-approximating the number of loop iterations needed for termination. We combine these *pieces* into piecewise-defined, lexicographic, or multiphase ranking functions.
The proposed technique has been implemented in SeaHorn – an LLVM based verification framework – targeting C code. Preliminary experimental evaluation demonstrated its effectiveness in synthesizing ranking functions and proving termination of C programs.

1 Introduction

The traditional method for proving program *termination* and other *liveness* properties is based on the synthesis of *ranking functions*, that is, for any potentially looping computation, proving that some well-founded metric strictly decreases every time around the loop.

State-of-the-art termination provers (e.g., [4,9,15]) reduce termination to the *safety* property that no program state is repeatedly visited (and it is not covered by the current termination argument), and compose termination arguments by repeatedly invoking ranking function synthesis tools (e.g., [7,3,25]).

In this work, we present a novel approach based on recent advances in *software model checking* to synthesize ranking functions and prove termination (and non-termination) of imperative programs. The core of our approach lies on an innovative use of *safety* verification techniques to build termination arguments. We use a safety verifier to systematically sample *terminating* program executions and extrapolate from these a candidate ranking function for the program, or to otherwise provide a witness for program non-termination. More specifically, rather than verifying that no program state is repeatedly visited, we verify the

* This material is based upon work funded and supported by NSF Award No. 1136008 the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002915

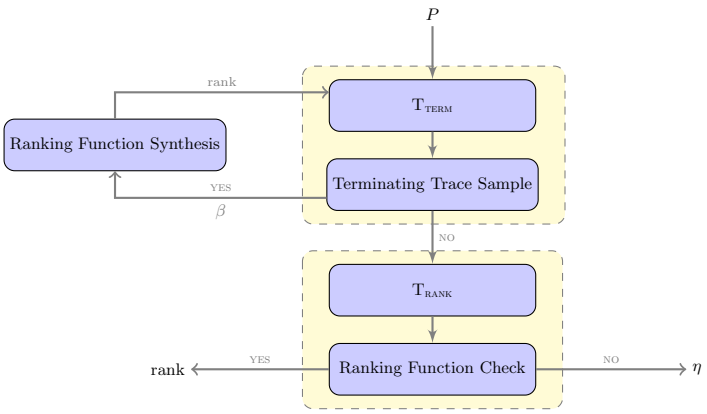


Fig. 1: Overview of our approach.

safety property that no program state is terminating (and it is not covered by the current termination argument). The counterexample traces are terminating program executions which provide an *under-approximation* of the terminating program states. From such traces we extrapolate a candidate ranking function by *over-approximating* the number of loop iterations to termination that is possibly valid also for other terminating program executions. The candidate ranking function can be an *affine* function, or a *piecewise-defined*, *lexicographic*, or *multi-phase* combination of affine functions. We then use the safety verifier to validate that the candidate ranking function is indeed a ranking function, or to provide a counterexample non-terminating program state.

The proposed approach has been implemented in SeaHorn [14] targeting C code. We show empirically that it performs well on a wide variety of benchmarks collected from SV-COMP 2015 [4]. In fact, it is competitive with state-of-the-art termination provers and is able to analyze programs that are out of the reach of existing techniques.

Overview. Figure 1 provides an overview of our approach for proving termination via safety verification. The overall algorithm is presented in Section 3.2. A program P systematically undergoes a transformation $\mathrm{T}_{\mathrm{TERM}}$ described in Section 4.1 which allows sampling terminating executions β not covered by the current candidate ranking function *rank*. The candidate *rank* is systematically refined as described in Section 4.2 until no β is left uncovered. Finally, P undergoes a final transformation $\mathrm{T}_{\mathrm{RANK}}$ described in Section 4.1 which allows validating the ranking function *rank* or providing a counterexample non-terminating state η.

[4] http://sv-comp.sosy-lab.org/2015/

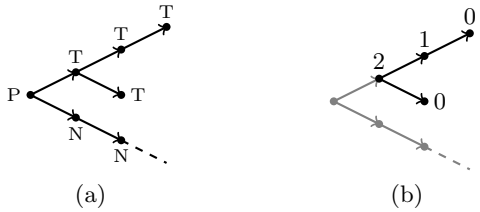


Fig. 2: Traces and Ranking Function.

2 Preliminaries

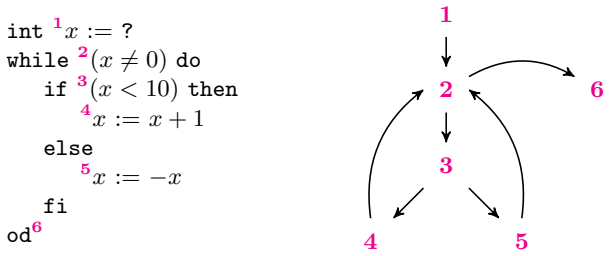
In this section, we introduce the basic concepts that serve in subsequent sections and we establish the notation used throughout the paper.

Transition Systems. We formalize programs using *transition systems* $\langle \Sigma, \tau \rangle$ where Σ is the set of program states and $\tau \subseteq \Sigma \times \Sigma$ defines the transition relation. Note that this model allows representing programs with (possibly unbounded) non-determinism. In the following, a program state $s \in \Sigma$ is a pair $\langle l, \bar{x} \rangle$ consisting of a program control point $l \in \mathcal{L}$ and a vector $\bar{x}$ of integers representing the values of the program variables at that control point. We write $\tau(s, s')$ for $\langle s, s' \rangle \in \tau$. The set of initial states is $\mathcal{I} \stackrel{\text{def}}{=} \{\langle i, \bar{x} \rangle \mid i \in \mathcal{L}\} \subseteq \Sigma$, where $i \in \mathcal{L}$ is the program initial control point, and the set of final states is $\mathcal{F} \stackrel{\text{def}}{=} \{\langle f, \bar{x} \rangle \mid f \in \mathcal{L}\} \subseteq \Sigma$, where $f \in \mathcal{L}$ is the program final control point.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of states in Σ determined by the transition relation τ, that is $\tau(s, s')$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. A state $s' \in \Sigma$ is *reachable* from another state $s \in \Sigma$ if and only if there exists a trace from s to s'. In the following, we write $\tau^*(s, s')$ to denote the existence of a trace from s to s'. A state $s' \in \Sigma$ is reachable if and only if it is reachable from an initial state $s \in \mathcal{I}$.

A state $s \in \Sigma$ is *terminating* if and only if all traces to which it belongs are finite, *potentially non-terminating* if and only if it belongs to at least one infinite trace. Dually, it is *non-terminating* if and only if all traces to which it belongs are infinite, and *potentially terminating* if and only if it belongs to at least one finite trace. Note that, terminating states are also potentially terminating states, and non-terminating states are also potentially non-terminating states. For instance, consider the traces depicted in Figure 2a: the states labeled with T are terminating, the states labeled with N are non-terminating, and the state labeled with P is potentially non-terminating and potentially terminating.

Ranking Functions. The traditional method for proving termination dates back to Turing [28] and Floyd [13] and it requires finding a *ranking function*, which is defined as follows:



(a) (b)

Fig. 3: Terminating program 3PIECES (a) and its control flow graph (b).

Definition 1 (Ranking Function). *Given a transition system* $\langle \Sigma, \tau \rangle$*, a* ranking function *is a partial function rank whose domain* $\mathrm{dom}(rank)$ *is a subset of the program states and whose value* (i) *strictly decreases through transitions between program states, that is* $\forall s, s' \in \mathrm{dom}(rank) : \tau(s, s') \Rightarrow rank(s') < rank(s)$*, and (ii) is bounded from below, that is* $\forall s \in \mathrm{dom}(rank) : rank(s) \geq 0$.

For instance, an obvious ranking function maps each program state to some well-chosen upper bound on the number of transitions until termination. Figure 2b shows a ranking function labeling the terminating states of Figure 2a.

Control Flow Graphs. The *control flow graph* (CFG) induced by a transition system $\langle \Sigma, \tau \rangle$ is a graph whose nodes are the program control points $\mathcal{L}$ and whose edges $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ are pairs of control points corresponding to transitions in the transition system: $\forall \langle l, \bar{x} \rangle, \langle l, \bar{x}' \rangle \in \Sigma : \tau(\langle l, \bar{x} \rangle, \langle l, \bar{x}' \rangle) \Rightarrow \langle l, l' \rangle \in \mathcal{E}$. In the following, we restrict our attention to *reducible* control flow graphs. A *loop* is a strongly connected component of the CFG with a single entry node h called *loop header*. The loops nested within a loop are the strongly connected components of the loop after removing the loop header. A loop *entry edge* is an edge whose source is outside the loop and whose target is inside the loop, a *loop edge* is an edge whose source and target are within the loop, and a loop *exit edge* is an edge whose source is inside the loop and whose target is outside the loop. Similarly, we can partition the corresponding transitions in the transition system into loop *entry transitions*, *loop transitions*, and loop *exit transition*.

Example 1. Consider the program in Figure 3a: the integer variable x is initialized non-deterministically; then, at each loop iteration, the value of x is increased by one or negated when it becomes greater than or equal to ten, until x becomes zero. The control flow graph of the program is depicted in Figure 3b. The program `while` loop corresponds to the strongly connected component of the CFG formed by the nodes **2**, **3**, **4** and **5**. The loop header is the node **2**. There is a single entry edge $\langle \mathbf{1}, \mathbf{2} \rangle$ and a single exit edge $\langle \mathbf{2}, \mathbf{6} \rangle$.

Remark 1. Note that it is not necessary for a ranking function to strictly decrease at each transition but only around each loop iteration [10]: $\forall\langle h, \bar{x}\rangle, \langle h, \bar{x}'\rangle \in \mathrm{dom}(rank) : \tau^*(\langle h, \bar{x}\rangle, \langle h, \bar{x}'\rangle) \Rightarrow rank(\langle h, \bar{x}'\rangle) < rank(\langle h, \bar{x}\rangle)$.

Example 2. The program 3PIECES of Figure 3a terminates whatever the initial value of the variable x. The following piecewise-defined function:

$$f(x) = \begin{cases} -x & x \leq 0 \\ 21 - x & 0 < x < 10 \\ x + 1 & 10 \leq x \end{cases}$$

is a valid ranking function for the program, which maps the initial value of x to the number of loop iterations needed for termination.

3 Verifying Termination via Safety

In the late 1970s, Lamport suggested a classification of program properties into the classes of *safety* and *liveness* properties [19]. Safety properties represent requirements that should be continuously maintained by the program. On the other hand, liveness properties represent requirements that need not hold continuously but whose eventual or repeated realization must be guaranteed. Thus, a counterexample to a safety property is a *finite* (prefix of a) program execution, while for a liveness property a counterexample is an *infinite* execution on which an event of interest does not occur. A prominent example of a liveness property is *termination*. Instead, *non-termination* is a safety property since any terminating (and, thus, finite) program execution is a witness against non-termination.

3.1 Verifying Safety Properties

The verification of safety properties often amounts to checking the reachability of an *error* location: *a program is safe when the error location is unreachable*. Otherwise the program is unsafe. In the former case, safety provers often provide an *invariant* testifying the validity of the property. In the latter case, safety provers usually provide a *counterexample* trace violating the safety property. In the following, we propose some examples to informally illustrate how safety properties can be verified by checking (un)-reachability of an error.

Verifying Non-Termination [5]. Consider the program in Figure 4a: the integer variables x and y are initialized with value zero and nine, respectively; then, at each iteration, x and y are increased by one, until x becomes equal to y. Since safety provers report counterexample traces reaching an error location, in order to verify that the program is non-terminating, we turn terminating traces into counterexamples to be found. In Figure 4b, we added an error location — defined as `assert(false)` — before the end of the program of Figure 4a: only terminating traces would execute `assert(false)`, thus the program is non-terminating since in this case the error location is in fact unreachable.

```
int [1]x := 0, y := 9
while [2](x ≠ y) do
    [3]x := x + 1
    [4]y := y + 1
od[5]
```

(a)

```
int [1]x := 0, y := 9
while [2](x ≠ y) do
    [3]x := x + 1
    [4]y := y + 1
od
assert (false)[5]
```

(b)

Fig. 4: Non-terminating program (a) annotated with an error location (b).

```
int [1]x := ?, r := max{−x, 21 − x, x + 1}
while [2](x ≠ 0) do
    r := r − 1
    assert (r ≥ 0)
    if [3](x < 10) then [4]x := x + 1 else [5]x := −x fi
od[6]
```

Fig. 5: Program 3PIECES annotated with a ranking function.

Verifying a Ranking Function. Safety provers can also be used to verify whether a given function is a ranking function for a program. For instance, to check wether $\texttt{max}\{-x, 21-x, x+1\}$ is a ranking function for the program 3PIECES shown in Figure 3a, we instrument the program as shown in Figure 5: we add a variable r initialized with the given function $\texttt{max}\{-x, 21-x, x+1\}$; then, within the loop, according to Definition 1 and Remark 1 (i) we strictly decrease the value of r (i.e., we decrease r by one), and (ii) we assert that the value of r is bounded from below (i.e., we assert that r is greater than or equal to zero). Note that the counterexample traces that would violate the assertion are either (prefixes of) non-terminating traces, or (prefixes of) traces that are terminating but require a higher number of loop iterations with respect to the initial value of r. In this case, since the assertion is never violated, the given function $\texttt{max}\{-x, 21-x, x+1\}$ is a valid ranking function for the program 3PIECES.

3.2 Verifying Termination via Safety

In the following, we describe the overall algorithm for proving termination via safety. We detail our specific implementation choices in Section 4.

The overall algorithm is illustrated by Algorithm 1. We verify termination of each loop in a program by implicitly constructing a lexicographic ranking function for nested sets of loops [1]. The function ISTERMINATING takes as input a transition system $\langle \Sigma, \tau \rangle$ and returns either TRUE: R, meaning that the program is terminating and R is a ranking function, or FALSE: ρ, meaning that the program is potentially non-terminating and ρ is a counterexample potentially

Algorithm 1 : Program Termination

```
1: function IsTerminating(⟨Σ, τ⟩)
2:     R ← ∅
3:     for h ∈ getLoops(⟨Σ, τ⟩) do                ▷ h is a loop header in the program
4:         r : ρ ← isLoopTerminating(h, ⟨Σ, τ⟩)
5:         if r then                                          ▷ the loop is terminating
6:             R ← R[h ↦ ρ]
7:         else return false : ρ                  ▷ ρ is a potentially non-terminating state
8:     return true : R                            ▷ R is a ranking function for the program
```

Algorithm 2 : Loop Termination

```
1: function isLoopTerminating(h,⟨Σ, τ⟩)                          ▷ h is the loop header
2:     rank ← 0                                  ▷ candidate ranking function initialization
3:     B ← ∅
4:     while true do
5:         β ← getTerminatingTrace(h, ⟨Σ, τ⟩, rank)
6:         if β then                          ▷ there are terminating traces violating rank
7:             B ← B ∪ β
8:             rank ← getCandidateRankingFunction(rank, B)
9:         else                                ▷ there are no terminating traces violating rank
10:            η ← isRankingFunction(rank)
11:            if η then                        ▷ η is a potentially non-terminating state
12:                return false : η
13:            else                              ▷ rank is a ranking function for the loop
14:                return true : rank
```

non-terminating initial state. Specifically, isTerminating invokes the function isLoopTerminating for each loop in the program (cf. Line 4) and maps each loop header h (cf. Line 3) to the returned ranking function (cf. Line 6), or returns as soon as a counterexample non-terminating state ρ is found (cf. Line 7).

The function isLoopTerminating is shown in Algorithm 2. Initially, isLoopTerminating assumes that all program states within the loop are non-terminating and looks for a counterexample, that is, a terminating trace β (cf. Line 5). Then, the call to the function getCandidateRankingFunction computes a candidate ranking function *rank* for the (potentially terminating) states along this trace (cf. Line 8). The original non-termination property is weakened to only search for terminating traces violating the candidate *rank*, and the process starts over. The information provided by the collected terminating traces is used to incrementally refine the candidate *rank* with further ranking function pieces. In case no further terminating traces violating *rank* are found (cf. Line 9), the call to the function isRankingFunction checks wether all program states within the loop are terminating (cf. Line 10): if so, *rank* is a ranking function for the loop (cf. Line 14); if not, a counterexample potentially non-terminating initial state η (that is, η belongs to at least one infinite trace) is returned (cf. Line 12). Note that isLoopTerminating might also not terminate (cf. Line 4).

4 Counterexample-Guided Ranking Function Synthesis

We now detail our implementation choices for the functions getTerminatingTrace and getCandidateRankingFunctions.

4.1 Search for Ranking Function Counterexamples

In Section 3.1, we have seen how to use a safety prover for verifying non-termination by turning terminating traces into counterexamples. In our approach, we use a similar intuition to systematically detect terminating traces violating a given candidate ranking function *rank*.

In the following, we consider a generic candidate *rank* and we introduce two program transformations T_{TERM} and T_{RANK} implemented by the functions getTerminatingTrace and isRankingFunction, respectively. We detail these transformations with respect to a specific candidate *rank* in the next Section 4.2.

T_{TERM} Transformation. Let h be a loop header within a program $\langle \Sigma, \tau \rangle$ and let *rank* be a candidate ranking function for the loop. We modify the program in order to turn terminating traces violating *rank* into counterexamples to be found. Specifically, we modify Σ in order to include the value of *rank* and we add an error state $\omega \notin \Sigma$: $(\Sigma \times \mathbb{Z}) \cup \{\omega\}$. In the following, s, s', and $\langle h, \bar{x} \rangle$ denote program states in Σ. We also define the modified transition relation τ as follows:

- for each loop *entry transition* $\tau(s, \langle h, \bar{x} \rangle)$ there exists an entry transition τ^{rank} which also includes the candidate *rank*:

$$\tau^{rank}(\langle s, r \rangle, \langle \langle h, \bar{x} \rangle, r' \rangle) \Leftrightarrow \tau(s, \langle h, \bar{x} \rangle) \wedge r' = rank(\bar{x})$$

- for each *loop transition* $\tau(\langle h, \bar{x} \rangle, s)$ whose source is the loop header h there exists a loop transition $\tau^{\ominus}$ which also strictly decreases the value of *rank*:

$$\tau^{\ominus}(\langle \langle h, \bar{x} \rangle, r \rangle, \langle s, r' \rangle) \Leftrightarrow \tau(\langle h, \bar{x} \rangle, s) \wedge r' = r \ominus 1$$

- for each loop *exit transition* $\tau(s, s')$ there exists transition $\tau^{\lhd}$ to the error state ω when the candidate ranking function is negative:

$$\tau^{\lhd}(\langle s, r \rangle, \omega) \stackrel{\text{def}}{=} r \lhd 0$$

For every other transition $\tau(s, s')$ there exists a transition $\tau'(\langle s, r \rangle, \langle s', r' \rangle) \Leftrightarrow \tau(s, s') \wedge r' = r$. The counterexample traces that reach the error state are traces that are leaving the considered loop but violate the candidate *rank* since they require a higher number of loop iterations with respect to the initial value of *rank*. The function getTerminatingTrace returns any of these counterexamples.

Theorem 1. *Let h be a loop header of a program $\langle \Sigma, \tau \rangle$ and let $\langle \Sigma', \tau' \rangle$ be the program resulting from the T_{TERM} transformation for a given candidate ranking function rank. Then, $\tau'^{*}(\langle \langle h, \bar{x} \rangle, rank(\bar{x}) \rangle, \langle s, r \rangle) \wedge \tau(\langle s, r \rangle, \omega)$ if and only if there exist $s' \in \Sigma$ $\tau(s, s')$ and the transition is an exit transition, and $\tau^{*}(s, s')$ and the trace visits the loop header h strictly more than $rank(\bar{x})$ times.*

```
int [1] x := ?, r := rank
while [2] (x ≠ 0) do
    r := r − 1
    if [3] (x < 10) then [4] x := x + 1 else [5] x := −x fi
od
assert (r ≥ 0)[6]
```

Fig. 6: Program 3PIECES annotated with a candidate ranking function *rank*.

Example 3. Consider again the program 3PIECES of Figure 3a. The transformation that we have just described intuitively corresponds to modifying 3PIECES as illustrated in Figure 6: we add a variable r initialized with the candidate *rank* within the entry transition $\langle \mathbf{1}, \mathbf{2} \rangle$; then, within the loop transition $\langle \mathbf{2}, \mathbf{3} \rangle$, we decrease the value of r by one and, after the loop, we assert that the value of r is greater than or equal to zero. The assertion is equivalent to adding an error transition $\langle \mathbf{2}, \boldsymbol{\omega} \rangle$ when r is negative. The counterexample traces that violate the assertion are traces that leave the loop after $rank - r$ loop iterations, where r is the (negative) value of the variable r after the loop.

$\mathrm{T}_{\mathrm{RANK}}$ Transformation. Note that traces that never leave the considered loop are not counterexamples since they never reach the error state. For this reason Algorithm 2 includes a final validation of the ranking function (cf. Lines 10-14). We implement this using an analogous program transformation: we define entry transitions τ^{rank} and loop transitions $\tau^{\ominus}$ as before:

$$\tau^{rank}(\langle s, r \rangle, \langle \langle h, \bar{x} \rangle, r' \rangle) \Leftrightarrow \tau(s, \langle h, \bar{x} \rangle) \wedge r' = rank(\bar{x})$$

$$\tau^{\ominus}(\langle \langle h, \bar{x} \rangle, r \rangle, \langle s, r' \rangle) \Leftrightarrow \tau(\langle h, \bar{x} \rangle, s) \wedge r' = r \ominus 1$$

unlike before, for each *loop transition* $\tau(s, s')$ we also define a transition $\tau^{\lhd}$ to the error state ω when the candidate ranking function is negative:

$$\tau^{\lhd}(\langle s, r \rangle, \omega) \stackrel{\text{def}}{=} r \lhd 0$$

Other transitions are again defined as $\tau'(\langle s, r \rangle, \langle s', r' \rangle) \stackrel{\text{def}}{=} \tau(s, s') \wedge r' = r$. The counterexample traces that violate the assertion are necessarily (prefixes of) non-terminating traces, since the $\mathrm{T}_{\mathrm{TERM}}$ transformation has excluded all terminating traces violating the candidate ranking function. The function isRANKINGFUNCTION returns the initial state of any of these counterexamples.

Theorem 2. *Let h be a loop header of a program $\langle \Sigma, \tau \rangle$ and let $\langle \Sigma', \tau' \rangle$ be the program resulting from the $\mathrm{T}_{\mathrm{RANK}}$ transformation for a given candidate ranking function rank. Then, $\tau'^{*}(\langle \langle h, \bar{x} \rangle, rank(\bar{x}) \rangle, \langle s, r \rangle) \wedge \tau(\langle s, r \rangle, \omega)$ if and only if $\tau^{*}(\langle h, \bar{x} \rangle, s)$ and the trace is the prefix of an infinite trace and visits the loop header h strictly more than $rank(\bar{x})$ times.*

Example 4. The transformation that we have just described intuitively corresponds to modifying the program 3PIECES of Figure 3a as illustrated in Figure 5.

4.2 Synthesis of Candidate Ranking Functions

The function GETCANDIDATERANKINGFUNCTION uses the terminating traces collected by GETTERMINATINGTRACE to extrapolate *affine* ranking function pieces which are combined into a candidate loop ranking function.

In Algorithm 2, the initial candidate is the constant function equal to zero (cf. Line 2). Then, the candidate ranking function is systematically updated in order to be valid for the newly discovered terminating traces, and possibly for other terminating traces not explicitly enumerated.

We extrapolate an affine ranking function piece from terminating traces mapping the initial states of these traces to the number of loop iterations needed for termination, and then finding an affine ranking function which fits these bits of information. More specifically, let $\{\langle \bar{x}_1, r_1\rangle, \langle \bar{x}_2, r_2\rangle, \dots\}$ be the set of pairs mapping the initial states $\bar{x}_1, \bar{x}_2, \dots$ of the collected terminating traces to the number $r_1, r_2, \dots$ of loop iterations needed for termination. We find a fitting affine function $\bar{m} \cdot \bar{x} + q$ of the program variables $\bar{x}$ by *linear interpolation*, that is by solving the system of equations:

$$\begin{aligned} \bar{m} \cdot \bar{x}_1 + q &= r_1 \\ \bar{m} \cdot \bar{x}_2 + q &= r_2 \\ &\vdots \end{aligned}$$

for the unknowns $\bar{m}$ and q.

Example 5. Let $\{\langle 9, 12\rangle, \langle 4, 17\rangle\}$ be the set of pairs mapping some initial states of the program 3PIECES of Figure 3a to the number of loop iterations needed for termination: the initial state with $x = 9$ needs 12 loop iterations, and the initial state with $x = 4$ needs 17 loop iterations. Solving the system of equations:

$$\begin{aligned} m \cdot 9 + q &= 12 \\ m \cdot 4 + q &= 17 \end{aligned}$$

yields the affine function $21 - x$ of the program variable x. Note that this is a valid ranking function for all initial states with $0 < x < 10$, and not only for the given initial states with $x = 9$ and $x = 4$ (cf. Example 2).

When the system is unsatisfiable, we discard all collected states and we start over by building a new ranking function piece. The ranking function pieces are alternatively combined either into *piecewise-defined*, *lexicographic*, or *multiphase* ranking functions [23].

Piecewise-Defined Ranking Functions. We represent piecewise-defined affine ranking functions using *max* combinations of affine ranking functions [24]:

$$\texttt{max}\{rank_1, \dots, rank_n\}$$

where $rank_1, \dots, rank_n$ are the affine ranking function pieces.

```
int [1]x := ?, y := ?, r := (x, y)
while [2](x > 0 ∧ y > 0) do
   if (snd(f) < 0) then r := (fst(r) − 1, y) else r := (fst(r), snd(r) − 1) fi
   assert (fst(r) ≥ 0)
   if [3](?) then [4]x := x − 1; [5]y := ? else [6]y := y − 1 fi
od[7]
```

Fig. 7: Program annotated with a lexicographic ranking function.

In the transformations $\mathrm{T}_{\mathrm{TERM}}$ and $\mathrm{T}_{\mathrm{RANK}}$ described in Section 4.1, the modified loop transitions $\tau^{\ominus}$ strictly decrease a *max* combination of ranking functions by strictly decreasing all its pieces:

$$\mathtt{max}\{r_1, \ldots, r_n\} \ominus 1 = \mathtt{max}\{r_1 - 1, \ldots, r_n - 1\}$$

In the added error transitions $\tau^{\lhd}$ a *max* combination of ranking functions is negative when all its pieces are negative:

$$\mathtt{max}\{r_1, \ldots, r_n\} \lhd 0 \Leftrightarrow r_1 < 0 \wedge \cdots \wedge r_n < 0$$

Example 6. The transformations $\mathrm{T}_{\mathrm{TERM}}$ and $\mathrm{T}_{\mathrm{RANK}}$ of the program 3PIECES of Figure 3a are shown in Figure 6 and Figure 5, respectively.

Lexicographic Ranking Functions. Lexicographic ranking functions are tuples:

$$(rank_1, \ldots, rank_n)$$

where $rank_1, \ldots, rank_n$ are affine ranking function pieces.

In the transformations $\mathrm{T}_{\mathrm{TERM}}$ and $\mathrm{T}_{\mathrm{RANK}}$, the modified loop transitions $\tau^{\ominus}$ strictly decrease a lexicographic ranking function resetting the less significant pieces to their initial affine expression:

$$(r_1, \ldots, r_i, r_{i+1}, \ldots, r_n) \ominus 1 = (r_1, \ldots, r_i - 1, rank_{i+1}, \ldots rank_n)$$

were $r_{i+1}, \ldots, r_n$ are negative and get reset to the initial $rank_{i+1}, \ldots, rank_n$. In the added error transitions $\tau^{\lhd}$ a lexicographic combination of ranking functions is negative when the first of its pieces is negative:

$$(r_1, \ldots, r_n) \lhd 0 \Leftrightarrow r_1 < 0$$

Example 7. Consider the program in Figure 7: the integer variables x and y are initialized non-deterministically; then, at each iteration, either the value of y is decreased by one or the value of x is decreased by one and the value of y is reset non-deterministically, until either variable is less than or equal to zero. The program terminates whatever the initial value of x and y. Let (x, y) be a candidate lexicographic ranking function for the program. In this case, the

transformation T_{RANK} intuitively corresponds to adding a variable r initialized with (x, y) within the entry transition $\langle \mathbf{1}, \mathbf{2} \rangle$; then, within the loop transition $\langle \mathbf{2}, \mathbf{3} \rangle$, decreasing the value of r lexicographically *resetting* its second component $\texttt{snd}(r)$ when negative, and asserting that its first component $\texttt{fst}(r)$ is greater than or equal to zero. The assertion is equivalent to adding an error transition $\langle \mathbf{2}, \boldsymbol{\omega} \rangle$ when $\texttt{fst}(r)$ is negative. In this case, since the assertion is never violated, (x, y) is a valid lexicographic ranking function for the program.

Multiphase Ranking Functions. Multiphase ranking functions specify ranking functions that proceed through a certain number of phases during program execution [23]. Those are represented as tuples:

$$(rank_1, \ldots, rank_n)$$

where $rank_1, \ldots, rank_n$ are affine ranking function pieces. Each piece represents a phase of the ranking function. In the transformations T_{TERM} and T_{RANK}, the modified loop transitions $\tau^{\ominus}$ strictly decrease a multiphase combination of ranking functions as follows:

$$(r_1, \ldots, r_i, r_{i+1}, \ldots, r_n) \ominus 1 = (r_1, \ldots, r_i - 1, r_{i+1}, \ldots r_n)$$

were $r_{i+1}, \ldots, r_n$ are negative (and, unlike in the lexicographic combination, are never reset). In the added error transitions $\tau^{\lhd}$ a multiphase combination of ranking functions is negative when the first of its pieces is negative:

$$(r_1, \ldots, r_n) \lhd 0 \Leftrightarrow r_1 < 0$$

In summary, our approach systematically collects terminating program executions and searches for a function that uniformly captures the termination argument of the program. The function can be an affine ranking function, or a piecewise, lexicographic, or multiphase combination of affine functions. Finally, we either manage to validate the candidate ranking function or provide a witness for program non-termination.

5 Implementation

Our approach is implemented in SeaHorn [5] – an LLVM [20] based safety verification framework. SeaHorn verifies user-supplied assertions as well as a number of built-in safety properties (e.g., buffer and signed integer overflows). It can also be used to check for inconsistent code in C programs [17].

SeaHorn is parameterized by the semantic representation of the program using Constrained Horn Clauses (CHCs), and by the verification engine that leverages the latest advances made in SMT-based Model Checking and Abstract Interpretation. Detailed information about SeaHorn can be found in [14]. The transformations T_{TERM} and T_{RANK} presented in Section 4.1 are used to enhance

[5] The tool can be downloaded from http://seahorn.github.io/

	Tot	Time
SeaHorn	135	1.71s
AProVE [27]	129	10.77s
FuncTion [29]	111	0.55s
HIPTnT+ [21]	152	0.62s
Ultimate [15]	109	8.45s

(a)

	SeaHorn			
	■	●	×	▲
AProVE [27]	39	33	96	22
FuncTion [29]	50	26	85	29
HIPTnT+ [21]	16	33	119	22
Ultimate [15]	55	29	80	26

(b)

Fig. 8: Overview of the experimental evaluation.

the CHCs passed to the verification engine. SeaHorn employs several SMT-based model checking engines based on PDR/IC3 [2], including Spacer [18]. The synthesis of candidate ranking functions presented in Section 4.2 uses Z3 [11] to find affine functions fitting the collected terminating states.

Experimental Evaluation. We compared our implementation in SeaHorn to the tools participated in the termination category of (SV-COMP 2015): AProVE [27], FuncTion [29], HIPTnT+ [21], and Ultimate Automizer [15]. We evaluated the tools against 190 terminating C programs collected from the SV-COMP 2015 benchmarks. Specifically, we selected only the programs that *all* tools could analyze (e.g., without parse errors or other clear issues) among the two most populated verification tasks of the termination category (i.e., crafted-lit and memory alloca). Note that other tools (e.g., FuncTion) provide a very limited support for arrays and pointers. Therefore, we were not able to analyze 30% of the considered benchmarks. The experiments were performed on a machine with a 2.90GHz 64-bit Dual-Core CPU (Intel i5-5287U) and 4GB of RAM, and running Ubuntu 14.04.

Figure 8 summarizes our experimental evaluation and Figure 9 shows a detailed comparison of SeaHorn against each other tool. In Figure 8a, the first column reports the total number of programs that each tool could prove terminating, and the second column reports the average running time in seconds for the programs where the tool proved termination. We used a time limit of 30 seconds for each program. In Figure 8b, the first column (■) lists the total number of programs that the tool was not able to prove termination for and that SeaHorn could prove terminating, the second column (●) reports the total number of programs that SeaHorn was not able to prove termination for and that the tool could prove terminating, and the last two columns report the total number of programs that both the tool and SeaHorn were able (×) or unable (▲) to prove terminating. The same symbols are used in Figure 9.

Figure 8a shows that SeaHorn is able to prove termination of 3.2% more programs than AProVE, 12.6% more programs than FuncTion, and 13.7% more programs than Ultimate Automizer. HIPTnT+ is able to prove termination of 8.9% more programs than SeaHorn, but SeaHorn can prove termi-

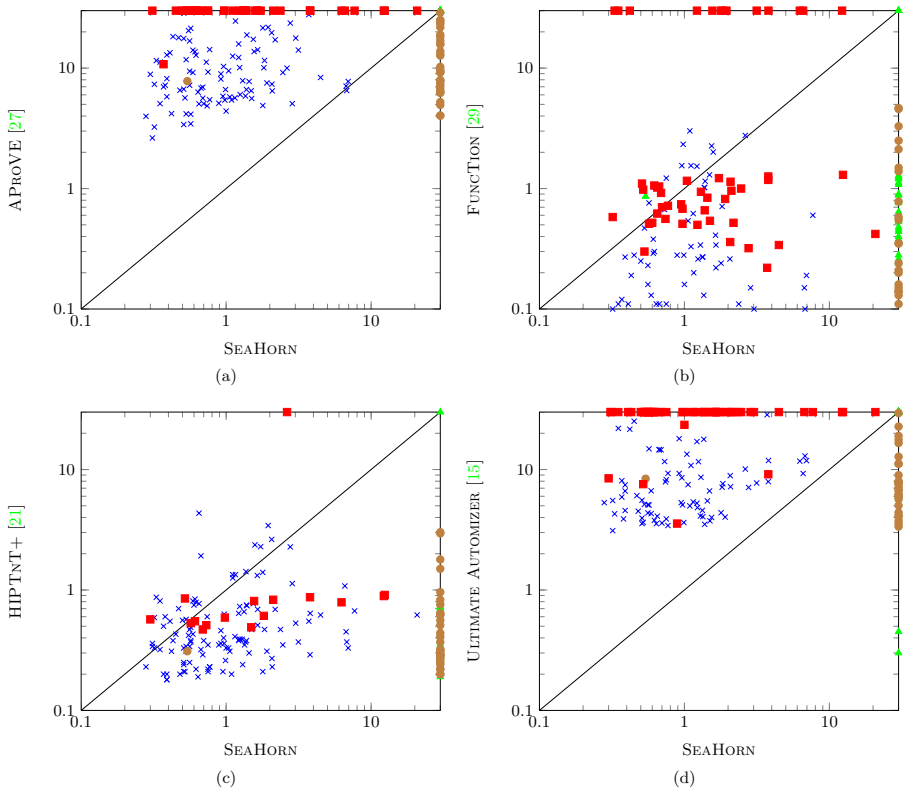


Fig. 9: Detailed comparison of SeaHorn against AProVE [27] (a), FuncTion [29] (b), HIPTnT+ [21] (c), and Ultimate Automizer [15] (d).

nation of 42.1% of the programs that HIPTnT+ is not able to prove terminating (8.4% of the total program test cases, cf. Figure 8b).

Figure 8b highlights the complementary strengths of SeaHorn and each of the other tools. Specifically, SeaHorn and AProVE seem to form the best combination with respectively 20.5% and 17.4% of the total program test cases that could be proved terminating only by one tool and not the other, and only 11.6% of the test case that could not be proved terminating by either tool.

Figure 9 shows that SeaHorn is generally faster than AProVE (cf. Figure 9a) and Ultimate Automizer (cf. Figure 9d), and often slower than FuncTion (cf. Figure 9b) and HIPTnT+ (cf. Figure 9c). In Figure 9b and Figure 9c, we also see that FuncTion and HIPTnT+ give up earlier when unable to prove termination, while SeaHorn, AProVE, and Ultimate Automizer usually persist with the analysis until the timeout (cf. also Figure 9a and Figure 9d).

Finally, we noticed that five of the *SV-COMP 2015* program test cases could be proved terminating only by SeaHorn (one only by AProVE, one only by

FuncTion, two only by HIPTnT+, and five only by Ultimate). No tool could prove termination of six of the program test cases.

6 Related Work

In the recent past, termination analysis has benefited from many research advances and powerful termination provers have emerged. Many approaches in this area reduce termination to a safety property. For instance, the approach implemented in Terminator [9] systematically verifies that no program state is repeatedly visited (and it is not covered by the current termination argument). The identified counterexamples are independently proved to be terminating [25] building a disjunctive well-founded termination argument [26]. A similar incremental approach is used in T2 [4] for the construction of lexicographic ranking functions. An automata-based incremental approach is described in [16] and implemented in Ultimate [15]. An approach based on conflict-driven learning is used in [12] to enhance the abstract interpretation-based termination analysis [30] implemented in FuncTion [29].

The incremental approach that we have proposed in this paper uses safety verifiers for proving termination in a fundamentally different way than existing methods: rather than systematically verifying that no program state is visited repeatedly, we systematically verify that no program state is terminating. Thus, our counterexamples are finite traces and do not need to be proven terminating.

The counterexample finite traces identified by our approach are used to extrapolate affine ranking functions. The linear interpolation that we use resembles the widening operator described in [30]. The extrapolated ranking functions are combined into a piecewise-defined, lexicographic, or multiphase ranking function for a program. Thus, our method provides more valuable information than just a positive or inconclusive answer like the methods based on the size-change termination principle [22] and implemented in AProVE [27], or like the already cited methods based on disjunctive well-foundedness and implemented in Terminator. Finally, compared to the incomplete methods implemented in AProVE and FuncTion, our method is also able to prove non-termination of program.

7 Conclusion and Future Work

This paper provides a new perspective on the use of safety verifiers for proving program (non-)termination. We have proposed a novel incremental approach, which uses a safety verifier to systematically sample *terminating* program executions and synthesize from these a ranking function for the program, or to otherwise provide a witness for program non-termination.

In the future we plan to adapt the approach in order to infer sufficient preconditions for program termination [6,30]. We also plan to extend the approach to other liveness properties [31,8].

References

1. A. M. Ben-Amram. Ranking Functions for Linear-Constraint Loops. In *VPT*, pages 1–8, 2013.
2. A. R. Bradley. IC3 and Beyond: Incremental, Inductive Verification. In *CAV*, page 4, 2012.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination analysis of integer linear loops. In *CONCUR 2005 - Concurrency Theory, 16th International Conference*, pages 488–502, 2005.
4. M. Brockschmidt, B. Cook, and C. Fuhs. Better Termination Proving through Cooperation. In *CAV*, pages 413–429, 2013.
5. H. Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Proving Nontermination via Safety. In *TACAS*, pages 156–171, 2014.
6. H. Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising Interprocedural Bit-Precise Termination Proofs. In *ASE*, 2015.
7. M. Colón and H. Sipma. Synthesis of linear ranking functions. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS*, pages 67–81, 2001.
8. B. Cook, H. Khlaaf, and N. Piterman. On Automation of CTL* Verification for Infinite-State Systems. In *CAV (I)*, pages 13–29, 2015.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *PLDI*, pages 415–426, 2006.
10. P. Cousot and R. Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.
11. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
12. V. D'Silva and C. Urban. Conflict-Driven Conditional Termination. In *CAV (II)*, pages 271–286, 2015.
13. R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
14. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The Seahorn Verification Framework. In *CAV I*, pages 343–361, 2015.
15. M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with Array Interpolation (Competition Contribution). In *TACAS*, pages 455–457, 2015.
16. M. Heizmann, J. Hoenicke, and A. Podelski. Software Model Checking for People Who Love Automata. In *CAV*, pages 36–52, 2013.
17. T. Kahsai, J. A. Navas, D. Jovanovic, and M. Schäf. Finding inconsistencies in programs with loops. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20*, pages –, to appear.
18. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, pages 17–34, 2014.
19. L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
20. C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–88, 2004.
21. T.-C. Le, S. Qin, and W.-N. Chin. Termination and Non-Termination Specification Inference. In *PLDI*, pages 489–498, 2015.
22. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL*, pages 81–92, 2001.

23. J. Leike and M. Heizmann. Ranking Templates for Linear Loops. In *TACAS*, pages 172–186, 2014.
24. S. Ovchinnikov. Max-Min Representation of Piecewise Linear Functions. *Contributions to Algebra and Geometry*, 42(1):297–302, 2002.
25. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.
26. A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.
27. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *TACAS*, pages 417–419, 2015.
28. A. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1948.
29. C. Urban. FuncTion: An Abstract Domain Functor for Termination (Competition Contribution). In *TACAS*, pages 464–466, 2015.
30. C. Urban and A. Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In *SAS*, pages 302–318, 2014.
31. C. Urban and A. Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In *VMCAI*, pages 190–208, 2015.