Efficient Ranking Function-Based Termination Analysis via Bidirectional Decompositional Search

Yasmin Chandini Sarita, Avaljot Singh, Shaurya Gomber, Gagandeep Singh, and Mahesh Vishwanathan

University of Illinois Urbana-Champaign, USA

Abstract. Synthesizing ranking functions is a common technique for proving the termination of loops. A ranking function must be bounded and decrease by a specified amount with each iteration for all reachable program states. Since the set of reachable states is unknown, loop invariants are used to over-approximate it, requiring the joint synthesis of ranking functions and invariants. Existing approaches either synthesize them independently, encode them into a single monolithic query, or connect them through ad hoc, one-way information flow, leading to inefficient exploration of large search spaces. We present Syndicate, a termination analysis framework based on the novel concept of *Bidirectional Decompositional Search (BDS)*. BDS keeps the ranking function and invariant synthesis decomposed but ensures continuous *bi-directional feedback*. This mutual guidance enables efficient exploration, significantly increasing the number of programs proven to terminate while reducing runtime compared to baselines without such feedback. Depending on the templates used, Syndicate is both relatively complete and provably efficient, outperforming existing techniques that achieve at most one of these guarantees. Despite its simplicity, Syndicate matches or surpasses state-of-the-art tools in termination proofs and runtime, demonstrating the effectiveness of bi-directional reasoning in termination analysis.
Github: `https://github.com/uiuc-focal-lab/Syndicate`

1 Introduction

Termination analysis is a key challenge in program verification, used in systems verification [9] and software model checking [20]. Existing approaches include reduction orders, recurrent sets, and fixpoint logic solving [11,14,17,22]. A widely used technique, and the focus of this work, is to prove termination by synthesizing *ranking functions*, which, for all reachable program states, are bounded below and strictly decrease with every program iteration. Since computing the exact set of reachable states is undecidable, loop invariants are used to over-approximate them. Thus, termination proofs require finding two unknowns: a ranking function and a loop invariant to prove the ranking function valid.

The appendix is available at `https://arxiv.org/abs/2404.05951`.

© The Author(s) 2026
R. Krebbers (Ed.): ESOP 2026, LNCS 16502, pp. 181–209, 2026 .
https://doi.org/10.1007/978-3-032-22723-2_7

In the context of classical property-guided synthesis [25,4], synthesizing both a ranking function and an invariant is challenging because when synthesizing either component, the other, which serves as its target property, is unavailable. For example, while synthesizing an invariant, without the ranking function, we cannot assess its effectiveness in proving termination. Similarly, synthesizing a ranking function without the supporting invariant is equally difficult. Thus, jointly finding both to prove termination poses the following challenges.

Challenge 1: Dual Search. At a high level, existing approaches address the dual search of ranking functions and invariants through one of the following strategies: (1) Methods like [10,17,14] synthesize the invariants and ranking functions independently, where the invariant synthesis does not know the current state of the ranking function search and vice versa. Due to the independence of the searches, these techniques frequently spend a significant amount of time discovering invariants that are either not useful to prove termination or are stronger than necessary. (2) Techniques like [9,24,7,27] use their current set of candidate ranking functions to guide a reachability analysis procedure. However, this procedure uses an external temporal safety checker, so insights gained during one call to the safety checker cannot be used to guide the ranking function generation procedure or future calls to the safety checker. (3) Methods such as [23,22,16] formulate both searches as a composite query. These techniques pose the synthesis of ranking functions and an over-approximation of the reachable states as the task of verifying the satisfiability of a combined formula. This couples both searches tightly, making the search space for the combined query significantly large, being a product of the individual search spaces.

Challenge 2: Efficiency & Completeness. Achieving efficiency and completeness while handling a wide variety of ranking functions and invariants is a significant challenge. While enumerating all possible candidate ranking functions and invariants can trivially ensure completeness, it often leads to prohibitively large runtimes, making such approaches impractical for real-world applications. Many existing techniques focus on specific classes of ranking functions [28,20,5], such as linear or polyhedral ones, providing completeness guarantees. However, this limits the kind of programs and ranking functions they can handle. While some methods offer completeness for a broader set of ranking functions [23,22], they still do not provide theoretical guarantees for efficient performance. Striking the right balance between efficiency and completeness while handling complex programs and ranking functions is a challenge in termination analysis.

Key Idea: Bidirectional Decompositional Search (BDS). The synthesis of ranking functions and invariants cannot be fully agnostic, as they depend on each other; yet encoding them together as a monolithic query is computationally expensive. *Bidirectional Decompositional Search* offers a principled middle ground, keeping the two searches decomposed yet allowing them to *mutually evolve* through continuous exchange of *bidirectional feedback*: (1) Counterexamples to the ranking function validation check that are not reachable program states can guide the invariant synthesizer to generate invariants that exclude these counterexamples. On the other hand, when the counterexamples are reach-

able and no invariant can exclude them, this search for possible invariants aids in discovering new conclusively reachable program states. (2) Provably reachable counterexamples found by an invariant synthesizer can guide the ranking function synthesizer to ensure that future potential ranking functions are bounded and reducing on these counterexamples. Together, (1) and (2) allow the two synthesis routines to interact *synergistically* through bidirectional feedback, enabling an efficient search for a ranking function and invariant just *strong enough* to prove termination. This approach remains efficient by maintaining smaller search spaces while leveraging feedback for faster convergence toward a termination proof. For programs with multiple loops, bidirectional feedback is even more crucial. Termination proofs across loops cannot be searched independently without losing precision, nor combined into a single query without incurring an intractable search space. In nested loops, the outer invariant over-approximates the initial states of the inner loop, while the inner invariant constrains the outer loop's body. Coordinating the searches for ranking functions and invariants across all loops through BDS leads to more efficient termination analysis.

Our framework: Syndicate. Based on our key idea, we introduce a general framework, Syndicate, that enables the synergistic synthesis of ranking functions and invariants for efficient termination analysis. It is parameterized by sets of possible invariants, $\mathcal{I}$, and ranking functions, $\mathcal{F}$, and can be instantiated with different choices to obtain different termination analyses. For each loop, counterexamples from the invariant search guide the ranking function search, and counterexamples from the ranking function search guide the invariant search. Further, the invariant and ranking function searches for each loop are guided by the analyses of the other loops in the program. Based on the choices for $\mathcal{I}$ and $\mathcal{F}$, Syndicate is relatively complete, meaning that if complete procedures exist for the template-specific functions in our framework, then Syndicate will either find a ranking function and an invariant within their respective templates to prove termination or will terminate, indicating that no such ranking function and invariant can be found. We show that Syndicate achieves this guarantee with greater efficiency than other approaches that lack completeness guarantees.

Main Contributions:

- We introduce the novel concept of *Bidirectional Decompositional Search (BDS)*, a principled framework that decomposes ranking-function and invariant synthesis into interdependent subproblems that exchange information through *bidirectional feedback* to enable efficient termination proofs.
- We present Syndicate, a general termination analysis framework that instantiates BDS and supports diverse templates for ranking functions and invariants, handling complex programs with nested and sequential loops, disjunctive conditionals, and non-linear statements.
- We establish an additive upper bound on the number of iterations in terms of the lattice depths of invariants and ranking functions, while guaranteeing relative completeness for a subset of templates (§ 4.3).
- We evaluate Syndicate on challenging benchmarks [2,12,10] (§ 6). By using BDS, Syndicate matches or surpasses state-of-the-art termination tools,

including those that employ SMT-level optimizations or techniques beyond ranking function synthesis [10,11,17,14,3]. It even proves the termination of some benchmarks that none of these advanced tools can prove.

2 Overview

Consider a loop program $\mathsf{L} = \texttt{while}(\beta)\ \texttt{do}\ \mathsf{P}_1\ \texttt{od}$ within a program P. Let $\mathcal{R}$ denote the set of reachable states at the loop entry and $[\![\mathsf{P}_1]\!]$ the transition relation capturing the semantics of the loop body, which may itself contain loops. A function f is a valid ranking function for L with respect to $\mathcal{R}$ and $[\![\mathsf{P}_1]\!]$ if it is bounded for all states in $\mathcal{R}$ and strictly decreases for all transitions in $[\![\mathsf{P}_1]\!]$, i.e., $\forall s \in \mathcal{R} : f(s) \geq \epsilon$ and $\forall (s, s') \in [\![\mathsf{P}_1]\!] : f(s) - f(s') \geq \delta$. In this work, we set $\epsilon = 0,\ \delta = 1$, though this can be generalized (Lemma 3, Appendix B).

Algorithm State. At each step of our algorithm, we maintain a state represented by a tuple $\langle t, \mathsf{A_L} \rangle$. The first component, t, is a set of pairs of states at the start and end of the loop body, serving as an under-approximation of the reachable states $\mathcal{R}$ and the loop body transition relation, i.e., $\mathsf{dom}(t) \subseteq \mathcal{R}$ and $t \subseteq [\![\mathsf{P}_1]\!]$. The second component, $\mathsf{A_L}$, stores the program along with invariants, I, for each loop, which provides an over-approximation of the reachable states and transition relations, i.e., $\mathcal{R} \subseteq \mathsf{I}$ and $[\![\mathsf{P}_1]\!] \subseteq \mathsf{I} \times (\mathsf{I} \cap [\![\neg\beta]\!])$. The set t can be initialized as an empty set or by executing the program on random inputs, while loop invariants are initialized to the trivial invariant, S (the set of all states, i.e., I_{true}). We define $\mathsf{A_L} = \mathsf{I}\ @\ \texttt{while}(\beta)\ \texttt{do}\ \mathsf{A}_1\ \texttt{od}$ as the *annotated program* corresponding to L. The semantics of L can be over-approximated by the annotated invariants in $\mathsf{A_L}$, denoted by $[\![\mathsf{A_L}]\!]$ (formally defined in § 3).

Syndicate Framework. We introduce Syndicate (Fig. 1), based on *Bidirectional Decompositional Search (BDS)* that interleaves *Generate* and *Refine* phases connected through *bidirectional* feedback. The Generate phase proposes a candidate ranking function, while the Refine phase refines the analysis state to better approximate the reachable states $\mathcal{R}$ and loop transition relation $[\![\mathsf{P}_1]\!]$. The Refine phase iteratively generates and checks new candidate invariants, but is different from typical counterexample guided synthesis because instead of running until a target property is proved, the refine phase guides the search for the target property, the ranking function. More concretely, after generating a candidate ranking function f in the Generate phase, if f is invalid, a counterexample p is obtained. Syndicate then invokes the Refine phase to generate and check new invariants, determining whether p is reachable and discovering other reachable states q in the process. If p is unreachable, I is refined to exclude it, while q still guides the next ranking function search. If p is reachable, both p and q guide the next Generate phase. Thus, the ranking function and invariant searches iteratively guide each other toward a termination proof. Syndicate is parameterized by $\mathcal{I}$, the set of invariants, and $\mathcal{F}$, the set of ranking functions. A counterexample s is deemed reachable when no valid invariant in $\mathcal{I}$ excludes it.

2.1 Syndicate Through an Example

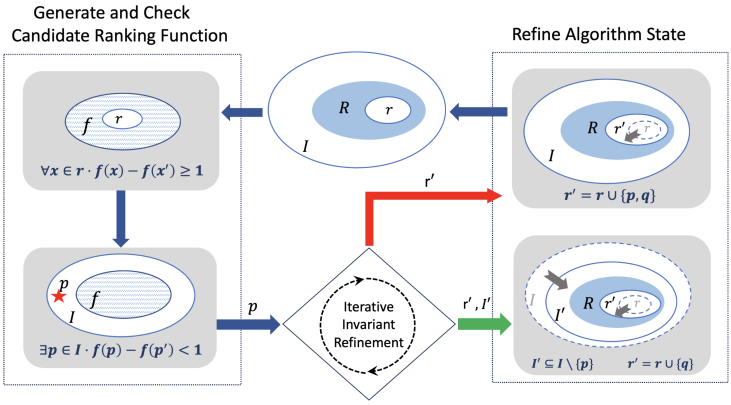


Fig. 1: For a single loop, we maintain r, I as canonical representations of $t, \mathsf{A_L}$ respectively. First, we use the set r to generate a candidate ranking function and then use the invariant I to check the validity. If we get a counterexample p, we try to refine the invariant I to exclude p. If p is not reachable, the invariant is refined successfully to I' (green arrow). Otherwise (red arrow), we add p to the set r. In both cases, the reachable states discovered while trying to refine I (denoted by q), are also added to the set r.

We demonstrate the behavior of SYNDICATE on a terminating program with a single loop, shown in Fig. 2. To simplify the presentation, we represent the analysis state as $\langle r, \mathsf{I}\rangle$ rather than $\langle t, \mathsf{A_L}\rangle$, where r denotes the loop-entry states and I the loop invariant. For a single loop, this representation is sufficient.

A ranking function for this loop is $\max(n - m + 1, 0) + \max(1 - m - y, 0)$ because for all reachable states, either $n - m + 1$ is reducing and $1 - m - y$ stays constant or $n - m + 1$ stays constant and $1 - m - y$ reduces from 1 to 0. However, we need an invariant $\mathsf{I} \equiv y + 1 = z$ to prove its validity (Fig. 3). We refer to the program in Fig. 2 as P, and the loop-body (lines 4-6) as P_1.

We generate random traces of the program and randomly initialize r with program states that are reachable at the loop head. Here, $r = \{(1, 2, 3, 4), (2, 2, 3, 4)\}$, where a program state is a tuple (m, y, z, n). I is initialized with the set of all states, i.e., $\mathsf{I} = \mathsf{true}$. SYNDICATE can handle other types of ranking functions and invariants, but for this example, let $\mathcal{F}$, the set of candidate ranking functions, be functions of the form $\max(a_0 + \sum_j a_j x_j, 0) + \max(b_0 + \sum_j b_j x_j, 0)$ and $\mathcal{I}$, the set of possible invariants, be conjunctions of constraints of the form $d_0 + \sum_j d_j x_j \geq 0$, where $a_j, b_j, d_j \in \mathbb{Z}$. hj find a valid ranking function and the corresponding invariant using separate queries while exchanging key information between the two searches, enabling an efficient termination analysis. The steps for our running example are explained below and also displayed in Table 1.

Iteration 1. *Generate Phase.* SYNDICATE begins by synthesizing a candidate ranking function f that decreases for all states in r. We solve for coefficients a_j, b_j in the query below, where $f \in \mathcal{F}$. Since all functions in $\mathcal{F}$ are bounded by 0, the

```
1  y = n;
2  z = n+1;
3  while (m+y >= 0 && m
       <= n){
4      m = 2 * m + y;
5      y = z;
6      z = z + 1;
7  }
```

Fig. 2: Example terminating program

$$
\begin{aligned}
& f(m', y', z', n') \\
& = \max\left(1 - (m' + y'), 0\right) + \max(n' - m' + 1, 0) \\
& = \max\left(1 - 2m - y - z, 0\right) + \max(n - 2m - y + 1, 0) \\
& = \max\left(1 - 2m - y - \boxed{(y+1)}, 0\right) + \\
& \qquad \max\left(n - m + 1 - (m + y), 0\right) \\
& = \max\left(n - m + 1 - (m + y), 0\right) \\
& \leq f(m, y, z, n) - 1
\end{aligned}
$$

Fig. 3: Value of Ranking function at the end of one iteration

boundedness condition need not be checked:

$$\forall (m, y, z, n) \in r \cdot \big(f(m, y, z, n) - f([\![\mathsf{P}_1]\!](m, y, z, n)) \geq 1\big).$$

We obtain $f(m, y, z, n) = \max(n - m + 1, 0)$ and check its validity w.r.t. $\mathsf{I} = \mathsf{true}$:

$$
\begin{aligned}
& \forall (m, y, z, n) \cdot \big((m + y \geq 0) \wedge (m \leq n)\big) \\
\implies & \big(f(m, y, z, n) - f([\![\mathsf{P}_1]\!](m, y, z, n)) \geq 1\big),
\end{aligned}
$$

where the guard $(m + y \geq 0) \wedge (m \leq n)$ comes from the loop condition. This yields a counterexample $p_1 = (1, -1, 2, 4)$.

Refine Phase. The algorithm now analyzes the counterexample p_1 to determine its reachability. There are two possibilities: if p_1 is reachable, the current ranking function is invalid; otherwise, the invariant must be strengthened to exclude p_1. To determine this, the algorithm proceeds through two subphases: invariant generation and invariant checking.

In the *invariant generation subphase*, the algorithm searches for a candidate I'' that includes all states in r but excludes p_1, solving for coefficients d_j:

$$(1, -1, 2, 4) \notin \mathsf{I}'' \ \wedge \ \forall (m, y, z, n) \in r : (m, y, z, n) \in \mathsf{I}''.$$

Solving yields $\mathsf{I}'' : y - z + 1 \geq 0$, and we set $\mathsf{I}' = \mathsf{I} \wedge \mathsf{I}''$. Next, in the *invariant checking subphase*, I' is verified against the initialization set $Init = \{(m, y, z, n) \mid (y = n) \wedge (z = n + 1)\}$, where $Init$ to refer to the initial states of the loop:

$$\forall s \in Init : s \in \mathsf{I}' \quad \wedge \quad \forall s \in \mathsf{I}' : [\![\mathsf{P}_1]\!]s \in \mathsf{I}'.$$

No counterexample is found, confirming I' as valid, and that p_1 is unreachable. The algorithm updates I to I', which is stored and reused in later iterations. Unlike prior tools [17,9,24] that invoke external solvers independently, SYNDICATE retains learned invariants across iterations, improving efficiency and convergence.

Iteration 2. As r remains unchanged, we check the validity of f with respect to the new invariant without generating a new ranking function. This check

	r	I	f	cex
1	$\{(1,2,3,4),(2,2,3,4)\}$	true	$\max(n-m,0)$	$(1,-1,2,4)$
2	$\{(1,2,3,4),(2,2,3,4)\}$	$y+1 \geq z$	$\max(n-m,0)$	$(1,-1,-1,1)$
3	$\{(1,2,3,4),(2,2,3,4),$ $(1,1,2,1)\}$	$(y+1 \geq z) \wedge (y+1 \leq z)$	$\max(n-m+1,0)$	$(1,-1,0,4)$
4	$\{(1,2,3,4),(2,2,3,4),$ $(1,1,2,1),(1,-1,0,4)\}$	$(y \geq z-1) \wedge (y \leq z-1)$	$\max(n-m+1,0)+$ $\max(1-m-y,0)$	-

Table 1: Steps in proof of termination by SYNDICATE for the program in Fig. 2

produces a new counterexample $p_2 = (1,-1,-1,1)$. In the invariant generation subphase, the algorithm attempts to strengthen the invariant $y+1 \geq z$ to exclude p_2, guided by both p_2 and the states in r. This process yields the candidate invariant $m-n-1 \geq 0$. In the subsequent checking subphase, the candidate is found invalid and produces a reachable counterexample $q_2^1 = (1,1,2,1) \in Init$. Since q_2^1 is reachable, it is added to r, improving the under-approximation of reachable states. q_2^1 satisfies $m = n$, a property absent in the previous elements of r. This observation later guides the generation of ranking functions that reduce on states where $m = n$. Because the new invariant is invalid, the algorithm returns to invariant generation with the additional constraint that all candidate invariants must include q_2^1. After further iterations, the refined invariant $\mathsf{I} = (y+1 \geq z) \wedge (y+1 \leq z)$ is obtained that successfully excludes p_2.

Iteration 3. As r has changed, we synthesize a new $f = \max(n-m+1,0)$. Checking against the current invariant produces $p_3 = (1,-1,0,4)$. Since no invariant in our template can exclude p_3, it must be reachable; we add it to r.

Iteration 4. With the updated r, SYNDICATE synthesizes $f = \max(n-m+1,0) + \max(1-m-y,0)$, which decreases for all states in r and the states that satisfy the invariant $(y+1 \geq z) \wedge (y+1 \leq z)$. The validity check succeeds, and this f is returned as the final ranking function proving termination.

BDS vs. Existing Methods. In iterations 1–2, counterexamples from ranking-function checks guide the search for stronger invariants, while in iterations 2–3, counterexamples from invariant checks guide the synthesis of valid ranking functions. This bidirectional feedback, central to the BDS principle, enables SYNDICATE to efficiently coordinate the two searches. Methods that rely on only one direction of feedback, such as passing the current ranking-function counterexample as a trace to an external safety checker [24,9,7], fail to reuse previously discovered reachable counterexamples or the prior state of the invariant search, making each call independent and redundant. In contrast, SYNDICATE leverages (p,p'), t, and previously learned invariants to navigate the search space effectively. To the best of our knowledge, no existing technique exploits intermediate counterexamples obtained during invariant search, such as q_2^1, to inform ranking-function synthesis. Fully combined approaches [3,23,22] must explore the *product* of the ranking-function and invariant search spaces, whereas SYNDICATE explores only their *sum*, yielding a smaller theoretical upper bound on the number of iterations (§ 4.3). As in § 6, SYNDICATE can prove more benchmarks terminating in less time than both unidirectional and fully coupled strategies.

2.2 Nested Loops

For nested loops, we have to store t, which contains pairs of states and serves as an under-approximation of the transition relation of the loop. Consider the program P in Fig. 4. P_i is the inner loop (lines 3-7), β_o and β_i are the loop guards at lines 2 and 3 respectively. Given a program state s before executing the inner loop body, we cannot compute the state s' just before line 8 because $[\![\mathsf{P}_i]\!]$ is also unknown. So, we use an invariant I_i to over-approximate $[\![\mathsf{P}_i]\!]$. Similarly, we use an invariant I_o for the outer loop. Note that the initial set for I_i depends on I_o. Conversely, I_o depends on $[\![\mathsf{P}_i]\!]$ which is approximated using I_i. So, the invariants corresponding to all the loops in a program depend on each other. Here, we represent our algorithm state as $\langle t, (\mathsf{I}_o, \mathsf{I}_i)\rangle$.

```
1  y = n; z = n+1;
2  while (y+n+1 >= k*k+z && y ==
       z+1){
3      while (m+y >= 0 && m <= n){
4          m = 2*m + y;
5          y = z;
6          z = z+1;
7      }
8      y++; z++; n--;
9  }
```

Fig. 4: Nested loop program

We use the same templates $\mathcal{F}$ and $\mathcal{I}$ for both loops, although this is not required by our algorithm. Both invariants are initialized as true, i.e., $\mathsf{I}_o = \mathsf{I}_i = \mathsf{true}$. Using this, we compute the initial set for the inner loop as $Init_i = \mathsf{I}_o \cap [\![\beta_o]\!] = \{(m, y, z, n) \mid (y + n + 1 \geq k^2 + z) \wedge (y = z + 1)\}$, and for the outer loop as $Init_o = \{(m, y, z, n) \mid (y = n) \wedge (z = n + 1)\}$. We start by analyzing the inner loop. Since its body is the same as the single-loop program in Fig. 2 and $Init_i$ is a subset of the corresponding $Init$, we can reuse the same iterations to obtain $f_i = \max(n-m+1, 0) + \max(1-m-y, 0)$, $\mathsf{I}_i \equiv (y+1 = z)$, and $\mathsf{I}_o = \mathsf{true}$. Next, Syndicate analyzes the outer loop. Similar to the single-loop case, we generate a candidate ranking function f. However, since the outer loop body contains a nested loop, we cannot symbolically execute it to obtain the state after one iteration. Instead, we introduce fresh variables $x'' = (m'', y'', z'', n'', k'')$ to represent the program state immediately after the inner loop terminates, assuming $x'' \in \mathsf{I}_i \wedge \neg\beta_i$. The ranking function validation query is defined as:

$$\mathsf{I}_o(m, y, z, n, k) \wedge \beta_o(m, y, z, n, k) \wedge \big(f(m, y, z, n, k) - f(m', y', z', n', k') < 1\big) \wedge \mathsf{I}_i(x'') \wedge \neg\beta_i(x''),$$
$$\text{where} \quad m' = m'',\ y' = y'' + 1,\ z' = z'' + 1,\ n' = n'' - 1,\ k' = k''.$$

We then refine the algorithm state. The validation query yields a counterexample (p, p'', p'), where p and p' denote the states at the start and end of one iteration of the outer loop, and p'' is the state at the exit of the inner loop. The refine phase can either (a) refine I_o to exclude p or (b) refine I_i to exclude p''. If at least one invariant is refined, we recheck the ranking function; otherwise, we add (p, p') to t. When refining an invariant, Syndicate iteratively performs the Generate and Check Invariant sub-phases. If a conclusively reachable state is discovered (e.g., a state in $Init_o$), it is reused to guide future ranking function synthesis.

	t	$(\mathsf{I}_o, \mathsf{I}_i)$	f	p, p'
1	$\{((2,2,3,4,1),(6,7,8,0,1)),$ $((2,2,3,5,1),(6,8,9,0,1))\}$	$(\mathsf{true}, y+1=z)$	$\max(n-k,0)$	$(1,2,4,5,5),(7,2,3,5,5)$
2	$\{((2,2,3,4,1),(6,7,8,0,1)),$ $((2,2,3,5,1),(6,8,9,0,1))\}$	$(y+1 \geq z, y+1=z)$	$\max(n-k,0)$	$(1,2,2,5,5),(8,3,4,5,5)$
3	$\{((2,2,3,4,1),(6,7,8,0,1)),$ $((2,2,3,5,1),(6,8,9,0,1))\}$	$((y+1 \geq z) \wedge (y+1 \leq z), y+1=z)$	$\max(n-k,0)$	$(1,2,3,5,5),(7,5,6,5,5)$
4	$\{((2,2,3,4,1),(6,7,8,0,1)),$ $((2,2,3,5,1),(6,8,9,0,1)),$ $((1,2,3,5,5),(7,5,6,5,5))\}$	$((y \geq z-1) \wedge (y \leq z-1), y+1=z)$	$\max(n-k+1,0)$	-

Table 2: Steps in proof of termination by SYNDICATE for the program in Fig. 4

If one invariant cannot be refined, the algorithm attempts to refine the other. Table 2 summarizes the steps for discovering the outer loop's ranking function. In § 4 and § 5, we generalize this procedure to nested loops of arbitrary depth.

2.3 Theoretical Guarantees

While SYNDICATE can be soundly instantiated with various templates, in § 4, we establish both relative completeness and efficiency guarantees if the template for ranking functions consists of a finite set of possibilities and the template for invariants is closed under intersection and does not have an infinite descending chain (with respect to the partial order defined in § 4). We further prove that even with a countably infinite ranking function template, SYNDICATE will be able to find a proof of termination if it exists within the templates.

SYNDICATE's bidirectional feedback improves efficiency by simultaneously guiding the search for ranking functions and invariants. Using a meet semilattice structure, we show that the worst-case bound for SYNDICATE is the sum of the depths of the ranking function and invariant lattices, a significant improvement over naive feedback methods, which would have a worst-case bound of the product of the size of the individual lattices. This improvement can be attributed to the synergy between the ranking function and invariant synthesis. The worst-case runtime analysis, detailed in § 4.3, shows that SYNDICATE maintains efficiency while providing theoretical guarantees that are not provided by state-of-the-art algorithms [10,17,14,11,3].

3 Annotated Programs

We illustrate our approach using programs written in a simple while-language:

$$\mathsf{P} ::= \texttt{skip} \mid x \leftarrow e \mid \mathsf{P};\mathsf{P} \mid \texttt{if } \beta \texttt{ then P else P} \mid \texttt{while } \beta \texttt{ do P od}.$$

Here, x denotes a program variable, e an arithmetic expression, and β a Boolean condition. We write $\mathsf{P}, \mathsf{P}_1, \ldots$ for programs, $\alpha, \alpha_1, \ldots$ for loop-free programs, and $\mathsf{L}, \mathsf{L}_1, \ldots$ for loops of the form `while` β `do P od`. A *program state* s assigns values to all variables of P, and S_P denotes the set of all such states. The semantics of a program, written $[\![\mathsf{P}]\!] \subseteq S_\mathsf{P} \times S_\mathsf{P}$, is the transition relation describing how execution transforms one state into another.As discussed in § 2.2, invariants

approximate reachable states for loops as well as initial states for nested or sequential loops. To capture this formally, we define *annotated programs*, where each loop is paired with an inductive invariant. These annotated programs are then used to define the validity of ranking functions and to introduce a partial order and a meet operation over annotated programs. Using these definitions, in § 4.3, we ensure that SYNDICATE refines the algorithm state in each iteration and is complete even for arbitrarily nested loops. Let $\mathcal{I} \subseteq 2^S$ be a collection of invariants closed under intersection. Annotated programs are defined as follows:

$$\mathsf{A} ::= \texttt{skip} \mid x \leftarrow e \mid \mathsf{A};\mathsf{A} \mid \texttt{if } \beta \texttt{ then A else A} \mid \mathsf{I}@\texttt{while } \beta \texttt{ do A od}$$

We use $\mathsf{A}, \mathsf{A}_1, \ldots$ to denote annotated programs. For an annotated program A, its *underlying program* is obtained by removing all loop annotations, written as $\mathcal{C}(\mathsf{A})$ (Definition 4, Appendix A). For a program P, the set of all annotated versions is $\mathsf{A}(\mathsf{P}) = \{\mathsf{A} \mid \mathcal{C}(\mathsf{A}) = \mathsf{P}\}$. We write A_L for the annotated subprogram of A corresponding to a loop L. An annotated program A is *correct* with respect to initial states $Init$, written $Init \models \mathsf{A}$, if all annotations are inductive invariants:

$$\begin{aligned}
Init \models \alpha &\triangleq \mathsf{true} \\
Init \models \texttt{if } \beta \texttt{ then } \mathsf{A}_1 \texttt{ else } \mathsf{A}_2 &\triangleq (Init \cap [\![\beta]\!] \models \mathsf{A}_1) \wedge (Init \cap [\![\neg\beta]\!] \models \mathsf{A}_2) \\
Init \models \mathsf{A}_1 \,;\, \mathsf{A}_2 &\triangleq (Init \models \mathsf{A}_1) \wedge (Init \circ [\![\mathsf{A}_1]\!] \models \mathsf{A}_2) \\
Init \models \mathsf{I} \,@\, \texttt{while } \beta \texttt{ do } \mathsf{A}_1 \texttt{ od} &\triangleq (Init \subseteq \mathsf{I}) \wedge ((\mathsf{I} \cap [\![\beta]\!]) \circ [\![\mathsf{A}_1]\!] \subseteq \mathsf{I} \times \mathsf{I}) \\
&\quad \wedge (\mathsf{I} \cap [\![\beta]\!] \models \mathsf{A}_1)
\end{aligned}$$

Next, we define the semantics of an annotated program A. We assume standard semantics for expressions and Boolean conditions: $[\![\alpha]\!]$ denotes the transition semantics of a loop-free program, and $[\![\beta]\!] \subseteq S$ the semantics of a Boolean expression. The semantics of annotated programs are defined inductively as:

$$\begin{aligned}
[\![\alpha]\!] &\triangleq [\![\alpha]\!] \\
[\![\texttt{if } \beta \texttt{ then } \mathsf{A}_1 \texttt{ else } \mathsf{A}_2]\!] &\triangleq \mathsf{id}_{[\![\beta]\!]} \circ [\![\mathsf{A}_1]\!] \cup \mathsf{id}_{[\![\neg\beta]\!]} \circ [\![\mathsf{A}_2]\!], \\
[\![\mathsf{A}_1; \mathsf{A}_2]\!] &\triangleq [\![\mathsf{A}_1]\!] \circ [\![\mathsf{A}_2]\!], \\
[\![\mathsf{I}@\texttt{while } \beta \texttt{ do } \mathsf{A}_1 \texttt{ od}]\!] &\triangleq \mathsf{I} \times (\mathsf{I} \cap [\![\neg\beta]\!]).
\end{aligned}$$

The key difference from unannotated semantics lies in the loop case: I overapproximates the reachable states at the loop head ($\mathcal{R} \subseteq \mathsf{I}$). We can now reason about when a ranking function proves termination of an annotated `while`-loop. Given an initial set of states, $Init \subseteq S$, a `while`-loop with loop guard β terminates if for every state $s \in Init$, there is a finite sequence of transitions starting from s and ending in s', such that the $s' \in [\![\neg\beta]\!]$.

Definition 1 (Correctness of a Ranking Function). *Let $Init \subseteq S$ be a set of initial program states and $f : S \to \mathbb{R}$ be a candidate ranking function. We say that f* establishes *the termination of annotated program $\mathsf{A}_\mathsf{L} = \mathsf{I} \,@\, \texttt{while } \beta \texttt{ do } \mathsf{A} \texttt{ od}$ from initial states Init, i.e., $Init \models \mathsf{A}_\mathsf{L}, f$ if*

– *The annotations of* $\mathsf{A_L}$ *are correct with respect to Init, i.e.,* $Init \models \mathsf{A_L}$*, and*
– *For all* $(s, s') \in \mathsf{id}_{\mathsf{I} \cap [\![\beta]\!]} \circ [\![\mathsf{A}]\!]$*,* $f(s) \geq 0$ *and* $f(s) - f(s') \geq 1$

If we can prove the correctness of a ranking function for every loop in an annotated program, then we have proven that the entire annotated program terminates. Since the invariants in an annotated program over-approximate the reachable states at each loop head, proving termination for the annotated program implies termination of the underlying program. In Definition 1, the ranking function f is required to be non-negative and decrease by at least 1 in each iteration. This can be generalized to arbitrary $\epsilon, \delta > 0$ (Lemma 3, Appendix B).

To reason about the completeness of Syndicate for arbitrarily nested loops, we define a partial order ($\preceq$) and a meet operation ($\wedge$) over annotated programs. For $\mathsf{A}_1, \mathsf{A}_2 \in \mathsf{A}(\mathsf{P})$, $\mathsf{A}_2 \preceq \mathsf{A}_1$ holds if, for each loop in P, the invariant in A_2 is contained in that of A_1 (formal definition in Appendix A). The meet $\mathsf{A}_1 \wedge \mathsf{A}_2$ annotates each loop with the intersection of their respective invariants. These notions yield the following result (proof in Appendix B):

Proposition 1. *Let* $\mathsf{A}_1, \mathsf{A}_2 \in \mathsf{A}(\mathsf{P})$ *be two annotated programs.*

1. *If* $Init \models \mathsf{A}_2$ *and* $Init \models \mathsf{A}_1$*, then* $Init \models \mathsf{A}_1 \wedge \mathsf{A}_2$*.*
2. *If* $Init \models \mathsf{A}_1, f$*,* $Init \models \mathsf{A}_2$*, and* $\mathsf{A}_2 \preceq \mathsf{A}_1$*, then* $Init \models \mathsf{A}_2, f$*.*

4 Syndicate Framework

The Syndicate framework leverages *Bidirectional Decompositional Search* to enable bidirectional synergy between ranking-function and invariant synthesis, allowing each to efficiently guide the other. Syndicate can be instantiated with *templates* that define the sets of possible ranking functions ($\mathcal{F}$) and invariants ($\mathcal{I}$), allowing each loop in a program to use a different template. We assume the set of all program states ($\mathsf{I}_{\mathsf{true}}$) is in $\mathcal{I}$, and that $\mathcal{I}$ is closed under intersection, i.e., if $\mathsf{I}_1, \mathsf{I}_2 \in \mathcal{I}$, then $\mathsf{I}_1 \cap \mathsf{I}_2 \in \mathcal{I}$. Given a terminating program P, the goal is to synthesize ranking functions from $\mathcal{F}$ and invariants from $\mathcal{I}$ that together prove the termination of every loop in P. To achieve this, we propose an algorithm parameterized by four sub-procedures: `getCandidate`, `check`, `getCex`, and `refine`. These sub-procedures can be instantiated using any implementation suited to the use case, as long as they adhere to the semantics in § 4.2. § 5 presents concrete instantiations satisfying these semantics, and § 4.3 establishes the algorithm's relative completeness and efficiency guarantees.

4.1 Algorithm State

Consider a scenario where Syndicate attempts to prove the termination of a loop $\mathsf{L} = \texttt{while}\ \beta\ \texttt{do}\ \mathsf{P}_1\ \texttt{od}$ within a program P, given initial states $Init$. Syndicate maintains an algorithm state used to generate and verify candidate ranking functions, refining it iteratively until convergence. This state consists of: (1) an annotated program $\mathsf{A} \in \mathsf{A}(\mathsf{P})$ such that $Init \models \mathsf{A}$, initialized by labeling

each loop in P with S (the set of all program states); and (2) a finite set $\mathsf{t} \subseteq S \times S$ such that for any correct annotation A' of P, $\mathsf{t} \subseteq \mathsf{id}_{\mathsf{I}' \cap [\![\beta]\!]} \circ [\![\mathsf{A}'[\mathsf{P}_1]]\!]$, where I' is the annotation of L in A'. The set t can be initialized by executing P on randomly chosen states from *Init* and collecting execution traces. Since each $(s, s') \in \mathsf{t}$ lies in the semantics of P_1, it can be used to infer a candidate ranking function from $\Pi(\mathsf{t}, \mathcal{F}) \triangleq \{f \in \mathcal{F} \mid \forall (s, s') \in \mathsf{t},\ f(s) \geq 0 \wedge f(s) - f(s') \geq 1\}$, the set of functions that are bounded and strictly decreasing over t. The annotated program is then used to check the validity of these candidate ranking functions.

Refinement of the Algorithm State. Checking the correctness of f may lead to at least one of the following cases. (1) The identification of additional pairs of states $(s_i, s_i') \notin t$ over which every valid ranking function must decrease. We add these newly discovered pairs to the algorithm state, i.e., $t' \leftarrow t \cup \{(s_i, s_i')\}$. As a consequence, $\Pi(\mathsf{t}', \mathcal{F}) \subset \Pi(\mathsf{t}, \mathcal{F})$. (2) The *refinement* of the annotated program to A' such that $\mathsf{A}' \prec \mathsf{A}$. Thus, the algorithm progresses either by eliminating additional potential ranking functions or constructing a more precise model for the semantics of the program by a progressive refinement of the annotations of P. Note that in our algorithm, the new annotation A' is always $\preceq \mathsf{A}$. An interesting and important observation at this point, described formally in Lemma 1, is that if there exists an annotated program A^* which can prove the validity of a ranking function f, then there also exists an annotated program $\mathsf{A}' \preceq \mathsf{A}$, which can prove f valid (Fig. 5). This implies that when we refine the program's annotations, although we narrow down the space of all possible annotations, this does not eliminate the possibility of verifying the validity of any correct ranking function. This fact allows Syndicate to descend the semi-lattice using any path and still make progress towards finding an annotated program with invariants that prove the validity of a ranking function. This is crucial in Syndicate's correctness and efficiency.

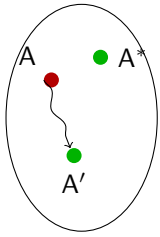


Fig. 5: Assume that A^* can prove the validity of f. Since $\mathsf{A}' \preceq \mathsf{A}$, A can be refined to A', which can also prove the validity of f.

Lemma 1. *Let* $\mathsf{A}, \mathsf{A}^* \in \mathsf{A}(\mathsf{P})$ *and* f *be a ranking function. If* $\mathit{Init} \models \mathsf{A}$ *and* $\mathit{Init} \models \mathsf{A}^*, f$ *then* $\exists \mathsf{A}' \preceq \mathsf{A}$ *and* $\mathit{Init} \models \mathsf{A}', f$.

Proof. Let $\mathsf{A}' = \mathsf{A} \wedge \mathsf{A}^*$. Then $\mathsf{A}' \preceq \mathsf{A}$, and by Proposition 1, $\mathit{Init} \models \mathsf{A}', f$.

4.2 Parameterized Algorithm

Syndicate (Algorithm 1) iteratively synthesizes a candidate ranking function and refines the algorithm state whenever the candidate is invalid. For programs with multiple loops, Algorithm 1 may be applied independently to each loop, in any order, to establish termination of the entire program. The algorithm is parameterized by four sub-procedures—`getCandidate`, `check`, `getCex`, and `refine`—as well as by templates for ranking functions ($\mathcal{F}$) and invariants ($\mathcal{I}$).

We require the following soundness conditions for the sub-procedures.

Definition 2 (Soundness of Sub-procedures). *The sub-procedures* `check`*,* `getCandidate`*, and* `getCex` *are sound if they satisfy the following:*

- `check` *returns* true *if and only if* $Init \models \mathsf{A}, f$ *holds for the given ranking function* f*.*
- `getCandidate` *returns a ranking function in* $\Pi(\mathsf{t}, \mathcal{F})$ *whenever this set is non-empty, and returns* false *otherwise.*
- *Whenever* `check` *returns* false*,* `getCex` *returns a counterexample satisfying conditions (i)–(iii) above.*

The procedure `getCandidate` (line 5) searches the space $\Pi(\mathsf{t}, \mathcal{F})$ for a ranking function compatible with the current algorithm state. If such a function exists, `getCandidate` returns one; otherwise, it reports failure. Given a candidate ranking function f, the procedure `check` (line 8) verifies whether f is valid with respect to the initial states and the current annotations, i.e., whether $Init \models \mathsf{A}, f$. If this check fails, `getCex` (line 10) produces a counterexample witnessing the failure.

Algorithm 1 SYNDICATE

```
1: procedure FIND_RANKING(t, A)
2:     refined ← false
3:     while true do
4:         if ¬refined then
5:             gen, f ← getCandidate(t, F)
6:             if ¬gen then
7:                 return false
8:         if check(f, A) then
9:             return true, f
10:        (p, p′) ← getCex(f, A)
11:        refined, A, t ← refine(A, (p, p′), t, I)
12:        if ¬refined then
13:            t ← t ∪ {(p, p′)}
```

A counterexample is a pair of states (p, p') such that (i) $(p, p') \in [\![\mathsf{A}[\mathsf{P}_1]]\!]$, (ii) $p \in \mathsf{I} \cap [\![\beta]\!]$, where I is the invariant labeling L in A, and (iii) the ranking condition is violated, i.e., either $f(p) < 0$ or $f(p) - f(p') < 1$. The counterexample is then used by `refine` to strengthen the algorithm state.

We now discuss the more involved `refine` sub-procedure in detail.

Refinement using `refine`. If f is not a valid ranking function for L then we get a counterexample (p, p') (line 10). This counterexample is then used to refine the program state (line 11). When P_1 is loop-free, the counterexample can be used in one of two ways: (1) The invariants in A are refined so that p is not an entry state of loop L or (p, p') is not in the semantics of the body P_1, or (2) a new ranking function is synthesized that behaves correctly on the pair (p, p').

When P_1 has loops, we define a sequence of states, $(p_0, p_1, \cdots, p_n)$ where $p_0 = p$, $p_n = p'$ and $p_1, \cdots, p_{n-1}$ are states occurring at the heads of the loops within P_1. We then try to refine the invariants to eliminate this sequence of states from the semantics of P_1. Importantly, even if this refinement does not eliminate the original counterexample (p, p'), it still makes progress by strengthening the invariant of at least one inner loop, thereby refining the algorithm state.

In both the case of loop-free body and body containing inner loops, the counterexample (p, p') is used to guide the refinement of A, demonstrating the *first direction of feedback*: ranking function search guiding invariant(s) search.

Thus, the crucial property required for the soundness of `refine` is as follows:

Definition 3 (Soundness of `refine`). *The procedure* `refine` *is sound if, for every annotated program* A*, counterexample* (p, p')*, parameter set* t*, and invariant template* $\mathcal{I}$*, the following conditions hold:*

1. *If* $\texttt{refine}(\mathsf{A}, (p, p'), \mathsf{t}, \mathcal{I})$ *returns* $(\mathsf{true}, \mathsf{A}', t')$*, then* $\mathsf{A}' \prec \mathsf{A}$ *and* $Init \models \mathsf{A}'$.
2. *If* $\texttt{refine}(\mathsf{A}, (p, p'), \mathsf{t}, \mathcal{I})$ *returns* $(\mathsf{false}, \mathsf{A}', t')$*, then* $\mathsf{A}' = \mathsf{A}$ *and, for every annotated program* A'' *such that* $Init \models \mathsf{A}''$ *and* $\mathsf{A}'' \prec \mathsf{A}$*, it holds that* $(p, p') \in \mathsf{id}_{\mathsf{I}'' \cap [\![\beta]\!]} \circ [\![\mathsf{A}''[\mathsf{P}_1]]\!]$ *where* I'' *is the invariant labeling* L *in* A''.

Implementing `refine` to satisfy these properties requires searching through the space of possible annotated programs, for example, using a CEGIS loop. As we will explain in § 5.1, this involves generating candidate invariants that exclude (p, p') and then checking their validity. While checking the validity of the generated invariants, `refine` can discover new reachable states and update t to t', including these states. Consequently, $t \subseteq t' \subseteq [\![\mathsf{P}_1]\!]$ and $\mathsf{dom}(t') \subseteq \mathcal{R}$. These newly discovered states in t' can then guide the ranking function search towards potentially correct ranking functions, demonstrating *the second direction of feedback*: invariant search guiding ranking function search.

Note that if `refine` is unable to refine (returns false), then by the definition of `refine`, for every $\mathsf{A}'' \prec \mathsf{A}$, (p, p') is in the semantics of the loop L captured by A''. The following lemma lifts this to all possible valid A'' and proves that if `refine` returns false, it means that (p, p') is in the semantics of the loop L captured by *all valid* annotated programs (not limited to $\prec \mathsf{A}$). So, we add (p, p') to t to get a new set that continues to satisfy the property that we required of t.

Lemma 2. *Suppose* $\texttt{refine}(\mathsf{A}, (p, p'), \mathsf{t}, \mathcal{I})$ *returns* $(\mathsf{false}, \mathsf{A}, t')$*. Consider any annotated program* $\mathsf{A}' \in \mathsf{A}(\mathsf{P})$ *such that* $Init \models \mathsf{A}'$*. Let* I' *be the annotation of* L *in* A'*. Then* $(p, p') \in \mathsf{id}_{\mathsf{I}' \cap [\![\beta]\!]} \circ [\![\mathsf{A}'[\mathsf{P}_1]]\!]$.

Proof. Suppose the claim is not true, i.e., for some correct annotated program A', $(p, p') \notin \mathsf{id}_{\mathsf{I}' \cap [\![\beta]\!]} \circ [\![\mathsf{A}'[\mathsf{P}_1]]\!]$. Consider the program $\mathsf{A}_1 = \mathsf{A} \wedge \mathsf{A}' \preceq \mathsf{A}$. As $Init \models \mathsf{A}$ and $Init \models \mathsf{A}'$, it follows from Proposition 1 that $Init \models \mathsf{A}_1$. Also, as $(p, p') \notin \mathsf{id}_{\mathsf{I}' \cap [\![\beta]\!]} \circ [\![\mathsf{A}'[\mathsf{P}_1]]\!]$ and $\mathsf{A}_1 \preceq \mathsf{A}'$, it follows from lemma 8 that $(p, p') \notin \mathsf{id}_{\mathsf{I}_1 \cap [\![\beta]\!]} \circ [\![\mathsf{A}_1[\mathsf{P}_1]]\!]$. So, `refine` can not return false as there is a correctly annotated program $\mathsf{A}_1 \preceq \mathsf{A}$ which does not contain (p, p'), leading to a contradiction.

In nutshell, by processing counterexample (p, p'), Syndicate always progresses. There are two cases to consider. If `refine` returns true then we have a new correct annotated program A' that refines A. On the other hand, if `refine` returns false then we have new set $\mathsf{t}_{\mathsf{new}}$ with the property that $\Pi(\mathsf{t}_{\mathsf{new}}, \mathcal{F}) \subset \Pi(\mathsf{t}, \mathcal{F})$. This is because $f \in \Pi(\mathsf{t}, \mathcal{F}) \setminus \Pi(\mathsf{t}_{\mathsf{new}}, \mathcal{F})$ since f is not a valid ranking function for the pair (p, p'). Thus, in this case $\Pi(\mathsf{t}_{\mathsf{new}}, \mathcal{F})$ becomes smaller.

4.3 Soundness, Completeness, and Efficiency of Syndicate

Achieving efficiency or completeness independently in termination analysis is relatively straightforward, and many techniques exhibit either one. However, combining these properties, along with soundness, into a single framework is a

challenge. While sound and complete algorithms can be devised by exhaustively exploring the search space, they tend to be inefficient due to the vast number of possibilities they need to explore. On the other hand, algorithms optimized for efficiency may sacrifice completeness or soundness by limiting the search space or relying on heuristics, potentially overlooking valid termination proofs. One of the contributions of this work is that SYNDICATE achieves all three properties by employing a bidirectional feedback mechanism. Algorithm 1 can be shown to be sound for proving the termination of loop L of P when instantiated with any ranking function and invariant templates if the four sub-procedures are sound using the Definitions 2, 3. In this context, soundness means that if our algorithm succeeds and returns a ranking function f from $\mathcal{F}$, then there exists an annotated program, A, using invariants in $\mathcal{I}$ such that $Init \models \mathsf{A}, f$.

Theorem 1 (Soundness). *Suppose there exist sound procedures for the functions* `getCandidate`, `check`, `getCex`, *and* `refine` *that satisfy the properties defined in § 4.2. Then if Algorithm 1 return* (true, f), *there exists* A *using invariants from* $\mathcal{I}$ *such that* $Init \models \mathsf{A}, f$.

Proof (Sketch). By the soundness of `refine`, the program state A remains correct ($Init \models \mathsf{A}$). Together with the soundness of `check`(f, A), this guarantees that any ranking function returned by the algorithm is correct.

Algorithm 1 can also be shown to be complete for a subset of the possible ranking function and invariant templates, meaning that if there is a function $f \in \mathcal{F}$ and an annotated program, A, using invariants from $\mathcal{I}$ such that f can be proved valid using A, then Algorithm 1 will prove termination, and if there does not exist such an f and A, then Algorithm 1 will terminate concluding that no ranking function and invariant can be found within the templates. The completeness of Algorithm 1 depends on the completeness of the subprocedures. So, we refer to it as relative completeness. By completeness of the subprocedures, we mean that each subprocedure must terminate and succeed whenever a valid answer exists: e.g., if a counterexample exists in the current iteration, `getCex` will find it; if a refinement is possible, `refine` will produce it.

Theorem 2 (Relative Completeness). *Suppose sound and complete procedures exist for* `getCandidate`, `check`, `getCex`, *and* `refine`, *specified in § 4.2. Assume* $\mathcal{F}$ *is finite, and* $\mathcal{I}$ *is closed under intersection with no infinite descending chains (under subset ordering). If there exist* $f \in \mathcal{F}$ *and* A *over* $\mathcal{I}$ *such that* $Init \models \mathsf{A}, f$, *then Algorithm 1 returns* (true, f'); *otherwise, it returns* false.

Proof (Sketch). Suppose there is a ranking function f^* and an annotated program A^* such that A^* is a correct annotation of P and f^* is a valid ranking function that proves the termination of L. Let (A, t) be the state of the algorithm at any point. Observe that because of the property that the algorithm maintains of t, we have $f^* \in \Pi(\mathsf{t}, \mathcal{F})$. By Lemma 1 there is a correct annotated program $\mathsf{A}'(= \mathsf{A} \wedge \mathsf{A}^*) \preceq \mathsf{A}$ such that f^* proves the termination of L in A'. Finally, the assumptions on $\mathcal{I}$ and $\mathcal{F}$ mean that $\Pi(\mathsf{t}, \mathcal{F})$ and $\mathsf{A} \downarrow = \{\mathsf{A}' \mid \mathsf{A}' \preceq \mathsf{A}\}$ are both

finite sets and hence there can only be finitely many steps of refinement. Thus Algorithm 1 will terminate with the right answer.

Ensuring Efficiency with Completeness. SYNDICATE's bidirectional feedback ensures efficiency, even in instantiations that have the completeness guarantees stated above. Consider a meet semilattice $(\mathcal{L}_\mathcal{F}, \preceq, \wedge)$ induced by $\mathcal{F}$ where each element is a set of ranking functions from $\mathcal{F}$ and $\preceq, \wedge$ are defined by the set operations $\subseteq, \cap$. In each refinement step, we either (1) add a pair of states to $\mathtt{t}$ creating $\mathtt{t}_{new}$, which decreases the size of the set of ranking functions Π determined by the algorithm's state, as $\Pi(\mathtt{t}_{new}, \mathcal{F}) \subset \Pi(\mathtt{t}, \mathcal{F})$. The number of times this refinement can be performed is bounded by the depth of the semilattice $\mathcal{L}_\mathcal{F}$, or (2) we refine the current annotation A to A', where $\mathsf{A}' \prec \mathsf{A}$. This is bounded by the depth of the meet semilattice for the annotated programs $\mathcal{L}_\mathsf{A}$. This bounds the number of iterations of our algorithm to $\mathsf{depth}(\mathcal{L}_\mathcal{F}) + \mathsf{depth}(\mathcal{L}_\mathsf{A})$.

This worst-case bound is much better than an unguided search for invariants and ranking functions. A naive termination algorithm that searches for invariants and ranking functions independently would realize a worst-case bound of $|\mathcal{L}_\mathcal{F}| \times |\mathcal{L}_\mathsf{A}|$, where $|\cdot|$ represents the number of elements in the lattice. Even a strategy that ensures refinement, a downward movement in the respective lattices, at each step of the algorithm, would have a worst-case bound of $\mathsf{depth}(\mathcal{L}_\mathcal{F}) \times \mathsf{depth}(\mathcal{L}_\mathsf{A})$. Thus, we argue that SYNDICATE's synergistic synthesis for invariants and ranking functions has significant merit in improving runtimes.

Infinite Ranking Function Templates. While the size of ranking function templates that satisfy the assumptions of Theorem 2 can be very large, it may be useful to consider infinite sets of ranking functions. When given an infinite, recursively enumerable ranking function template, if the assumptions about the sub-procedures and the invariant template from Theorem 2 hold, we can still ensure that SYNDICATE can find a proof of termination if such a ranking function and invariants exist in their respective templates. For example, if $\mathcal{F}$ is the set of all linear ranking functions with integer coefficients, we can define a sequence of finite sets such that for all $f \in \mathcal{F}$, f is in at least one of the finite sets. In this case, we can define sets $\mathcal{F}^{10}, \mathcal{F}^{100}, \mathcal{F}^{1000}, \cdots$, where $\mathcal{F}^n$ refers to the subset of $\mathcal{F}$ where the absolute value of each coefficient is bounded by n. Each of these sets is finite and every ranking function in $\mathcal{F}$ is included in at least one of the sets. We can then call Algorithm 1 on each set $\mathcal{F}^n$. Since each $\mathcal{F}^n$ is finite, Algorithm 1 will terminate for each set, either finding a ranking function and invariant or proving that none exist within their templates. This leads to an algorithm that will always find a ranking function and annotated program that proves termination if they exist given $\mathcal{F}$ and $\mathcal{I}$. This result holds for recursively enumerable ranking function templates because we can define a series of finite sets that include every ranking function in the template.

5 Instantiation of SYNDICATE

In this section, we describe one instantiation of SYNDICATE. § 5.1 outlines the sound modeling of the four core sub-procedures: `getCandidate`, `check`, `getCex`,

and `refine`. These sub-procedures, depending on the templates and programs analyzed, can yield relatively complete termination algorithms, as discussed in § 4.3. In § 5.2, we introduce new parameters specific to this instantiation that balance the exploration of the ranking function and invariant search spaces. Tuning these parameters improves the algorithm's efficiency in practice.

5.1 Modeling sub-procedures using SMT

In this instantiation of SYNDICATE, we use SMT queries to check the soundness of ranking functions for a given loop. We define $\mathcal{F}$ as a template for ranking functions. For the function `getCandidate`, given t, we generate an SMT query encoding a symbolic ranking function in $\mathcal{F}$ that satisfies the reducing and bounded conditions for all of the pairs of states in t. Finding a satisfying assignment to this query is equivalent to finding a candidate ranking function. If there is no satisfying assignment, then $\Pi(\mathsf{t}, \mathcal{F})$ is empty, and `getCandidate` returns false.

We can also use an SMT query for a function that implements the semantics of both `check` and `getCex`. `check` checks the validity of f given A. First, we define variables representing the state at the start of one iteration of the loop, $(x_1, \cdots, x_j)$, and add the condition $(x_1, \cdots, x_j) \in \mathsf{I}$ where I is the invariant of the loop we are analyzing, retrieved from A. Then, we encode the over-approximation of the loop body defined by A into the SMT query. For each loop in the body of the outer loop, we define new variables, $(x_{i,1}, \cdots, x_{i,j})$ to represent the states at the exit of each inner loop, adding the condition $(x_{i,1}, \cdots, x_{i,j}) \in \mathsf{I}_i \cap [\![\neg \beta_i]\!]$, where I_i and β_i are the invariant and loop guard of the inner loop from A. We define variables representing the state at the end of the iteration, $(x'_1, \cdots, x'_j)$, and set these variables equal to the output state of $[\![\mathsf{A}]\!]$ when the input state is $(x_1, \cdots, x_j)$. Checking the validity of f using A is reduced to determining the satisfiability of the formula: $(f(x_1, \cdots, x_j) - f(x'_1, \cdots, x'_j) < 1) \vee (f(x_1, \cdots, x_j) < 0)$. We return true if this formula is unsatisfiable, and false otherwise. In Algorithm 1, `getCex` is only called when `check` returns false. In this case, from the SMT query in `check`, we have a satisfying assignment to $(x_1, \cdots, x_j)$ and $(x'_1, \cdots, x'j)$. Since we defined `check` to define symbolic states,$(x_{i,1}, \cdots, x_{i,j})$, at the end of each inner loop, instead of only returning (p, p'), `check` returns $(p_1, \cdots, p_m)$, where $p_1 = p$, $p_m = p'$ and $p_i = (x_{i,1}, \cdots, x_{i,j})$ for all $i \in [1, m]$.

Algorithm 2 Algorithm for `refine`

```
1: procedure refine(A, (p, p'), t, I)
2:     inv_c = { }
3:     A', gen ← getInv(t, (p, p'), inv_c)
4:     while gen do
5:         if checkInv(A') then
6:             return true, A ∧ A', t
7:         (c, c'), for_pre ← getCexInv(A')
8:         if for_pre then
9:             t ← t ∪ {(c, c')}
10:        else
11:            inv_c ← inv_c ∪ {(c, c')}
12:        A', gen ← getInv(t, (p, p'), inv_c, I)
13:    return false, A, t
```

Implementation of `refine`. We present an algorithm implementing the semantics of `refine` from § 4.2, starting with a single-loop program. Given a

candidate ranking function f and a counterexample (p, p'), the algorithm state contains an annotated program with invariant I. We aim to refine I to $\mathsf{I}' \subseteq \mathsf{I} \setminus \{p\}$ by finding a valid invariant I'' that excludes p while including all states s such that $(s, s') \in t$. If such I'' exists, we return $\mathsf{I} \cap \mathsf{I}''$, which lies in $\mathcal{I}$ since it is closed under intersection and satisfies $\mathsf{I} \cap \mathsf{I}'' \subset \mathsf{I}$. To find I'', we iteratively generate and validate candidate invariants until either a valid one excluding p is found or none exists. Algorithm 2 outlines this process: lines 3 and 12 generate candidates, line 5 checks validity, and lines 7-11 update the information for subsequent iterations. We maintain inv_c, the set of counterexample pairs from previous iterations, initialized as {} (line 2). Candidate invariants (`getInv`) and their validity (`checkInv`) are checked via SMT queries.

$$\Big(p \notin \mathsf{I}''\Big) \wedge \Big(\bigwedge_{(s,s') \in t} s \in \mathsf{I}''\Big) \wedge \Big(\bigwedge_{(s,s') \in inv_c} s \in \mathsf{I}'' \implies s' \in \mathsf{I}''\Big) \tag{1}$$

$$\Big(s \in Init \wedge s \notin \mathsf{I}''\Big) \vee \Big(s \in \mathsf{I}'' \wedge [\![\mathsf{P}]\!] s \notin I''\Big) \tag{2}$$

If the SMT query checking the validity of I'' is satisfiable, `getCexInv` returns a pair of states (c, c'), s.t. either $c \in Init \wedge (c, c') \notin \mathsf{I}''$ or $c \in \mathsf{I}'' \wedge c' \notin I''$. In the former case, we add this state to t (line 10). In the latter, we add it to inv_c (line 11). With updated t and inv_c, we repeat the process of generating and checking a new possible invariant. When `refine` returns the updated t, the counterexamples from the invariant search guide the ranking function search.

Multiple Loops. For multiple loops, we need to find $\mathsf{A}' \prec \mathsf{A}$, where A is the annotated loop program in our algorithm state. Here, `check` returns $(p_1, \cdots, p_m)$ instead of only (p, p'). So, we have counterexamples, p_i, for each invariant, I_i, in the program. For each invariant, we maintain the set t_i and inv_{c_i}. t_i is initialized from the initial program execution traces. inv_{c_i} is initialized to an empty set. We use an SMT query to generate candidate invariants, I'_i for all of the loops such that at least one of the loops in A that excludes the counterexample, p_i associated with I_i. Formally, we find a satisfying assignment to the constants in I'_i such that

$$\bigvee_i \Big(\neg p_i \in \mathsf{I}_i \wedge \bigwedge_{(s,s') \in t_i} s \in \mathsf{I}_i \wedge \Big(\bigwedge_{(s,s') \in inv_{c_i}} s \in \mathsf{I}_i \implies s' \in \mathsf{I}_i\Big)\Big) \tag{3}$$

Once candidate invariants are generated for all loops, if at least one I'_i excludes its p_i, we validate them as in the single-loop case, adding counterexamples to t_i or inv_{c_i} accordingly. We repeat this process until at least one invariant is refined or no valid invariant exists for any loop that excludes its p_i. In the latter case, we have shown that there exists a sequence of reachable states, with respect to the invariant templates, that transitions from p to p'. Hence, (p, p') lies in the semantics of the loop body for all possible refinements of A, and we return false. For the sub-procedures to be complete, the set of ranking functions in $\mathcal{F}$, their boundedness and decrease conditions, and both the invariant generation and validation queries must be expressible in a decidable SMT theory.

5.2 Adaptive Exploration for Efficient Implementation

Although Algorithm 1 is sound and complete with appropriate templates, it can be inefficient in practice. To improve efficiency, we introduce two parameters: (1) $\mathtt{P}_{\boldsymbol{ref}}$, the maximum number of times `refine` is called for a given candidate ranking function, and (2) $\mathtt{P}_{\boldsymbol{iter}}$, the maximum number of iterations within `refine`. These parameters prevent Syndicate from spending excessive time attempting to validate an invalid ranking function or prove an unreachable state unreachable. Once either limit is reached, Syndicate stops refining the invariant. The modified pseudocode is provided in Appendix E. Users can tune $\mathtt{P}_{\boldsymbol{ref}}$ and $\mathtt{P}_{\boldsymbol{iter}}$ to balance exploration between invariant and ranking-function searches.

When encountering a counterexample (p, p') for the ranking function, if `refine` is not invoked or cannot fully explore the invariant search space, we cannot conclude that (p, p') is reachable. In such cases, (p, p') is added to a separate set $\mathsf{t}^?$, containing pairs whose reachability is inconclusive. During subsequent invariant refinements, Syndicate checks whether A can prove any pairs in $\mathsf{t}^?$ unreachable and removes them accordingly. Candidate ranking functions are then generated using $\mathsf{t} \cup \mathsf{t}^?$, ensuring that each candidate $f \in \mathcal{F}$ decreases and is bounded over these pairs. If no candidate is found, the algorithm terminates. Since $\mathsf{t}^?$ may include unreachable pairs, this may cause early termination even when a valid ranking function exists. To retain completeness, if no valid ranking function is found, we rerun Syndicate with $\mathtt{P}_{\boldsymbol{ref}}, \mathtt{P}_{\boldsymbol{iter}} = \infty$, reusing the last t and A, thereby combining efficiency with completeness guarantees.

6 Evaluation

Implementation Details. We implemented Syndicate using the instantiation and parameters introduced in Section 5. For ranking functions, we adopt a commonly used template of lexicographic functions $\langle e_1, \ldots, e_n \rangle$, where each $e_k = \sum_i \max(a_0^i + \sum_j a_j^i x_j, 0)$, and x_j represents program variables with coefficients $a_j^i \in \mathbb{Z}$. This form generalizes lexicographic, linear, and non-linear ranking functions. We denote by $T(i, n)$ a template with i summands per lexicographic component and n components in total. Although our implementation supports any $i, n \in \mathbb{Z}$, we use five representative templates in the evaluation: $\mathcal{F}_1 = T(1, 1)$, $\mathcal{F}_2 = T(1, 2)$, $\mathcal{F}_3 = T(1, 3)$, $\mathcal{F}_4 = T(2, 1)$, and $\mathcal{F}_5 = T(2, 2)$. Invariants are represented as conjunctions of linear constraints $d_0 + \sum_j d_j x_j \geq 0$, where $d_j \in \mathbb{Z}$.

The implementation supports integer programs and function calls to random number generators. It can be directly extended to handle other arithmetic types supported by SMT solvers and non-recursive functions through inlining. For programs with multiple loops, since Equation 3 can yield large SMT queries, we refine invariants per loop whenever possible using smaller SMT queries that exclude the state p_i from I_i. If refinement fails, we add (p, p') to $\mathsf{t}^?$ as described in Section 5.2. We use a timeout of 120s, counting timed-out instances as 120s when computing averages. We implemented Syndicate in Python and used the Z3-Java API to validate ranking functions and invariants. All experiments were run on a 2.50 GHz 16-core Intel i9-11900H CPU with 64 GB of RAM.

Template	B-synergy1	B-synergy2	SYNDICATE
$\mathcal{F}_1$	76	84	**98**
$\mathcal{F}_2$	125	128	**142**
$\mathcal{F}_3$	132	127	**148**
$\mathcal{F}_4$	110	95	**128**
$\mathcal{F}_5$	115	113	**134**

Table 3: Benchmarks (out of 168) proved by SYNDICATE and the two unidirectional baselines across different templates.

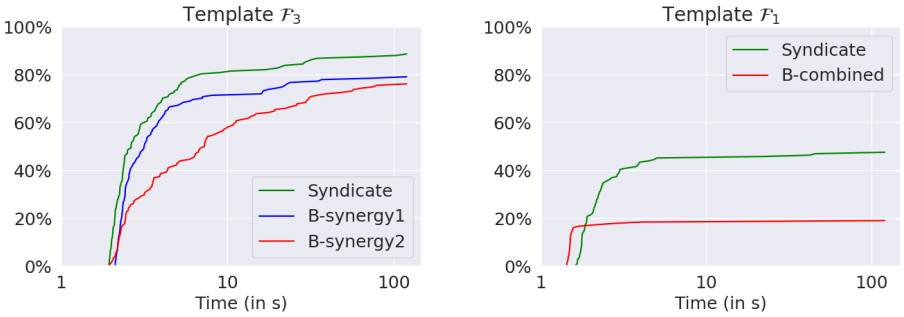
Benchmarks. We evaluate SYNDICATE on 168 integer-program benchmarks collected from νTerm [10], including all terminating programs in the *Term-crafted* set of SV-COMP [2], the *AProVE_09* set from TermComp [12], and the ν*Term-advantage* suite [10]. These standard benchmarks, originally in C and translated to Java by [10], include non-linear assignments and guards, non-determinism, and nested or sequential loops of depth up to three. Integer programs form a strong benchmark class since they over-approximate general (even heap-based) program semantics [1,18,21], making termination of such abstractions a standard evaluation metric [14,17,10,3,28,24,22].

Evaluation. Section 6.1 evaluates the benefits of SYNDICATE 's bidirectional decompositional search, where bidirectional feedback creates synergy between the independent ranking-function and invariant searches. We compare against baselines that (1) employ feedback in only one direction [24,9,7,27], or (2) jointly synthesize both using a monolithic query [3,22,23]. We also evaluate multiple algorithmic variants of SYNDICATE and analyze the causes of failure.

In Section 6.2, we also compare SYNDICATE with state-of-the-art termination tools [14,11,10,3,17], which use sophisticated techniques such as neural networks, multiple ranking functions, reduction orders, and SMT-specific optimizations. Despite not using these advanced techniques, SYNDICATE achieves comparable or better performance, primarily due to its novel bidirectional feedback. These techniques are orthogonal and can be integrated with SYNDICATE to further improve scalability and precision. Finally, Section 6.3 presents two representative benchmarks where SYNDICATE proves termination quickly, where other techniques fail, including one that none of the existing tools can handle.

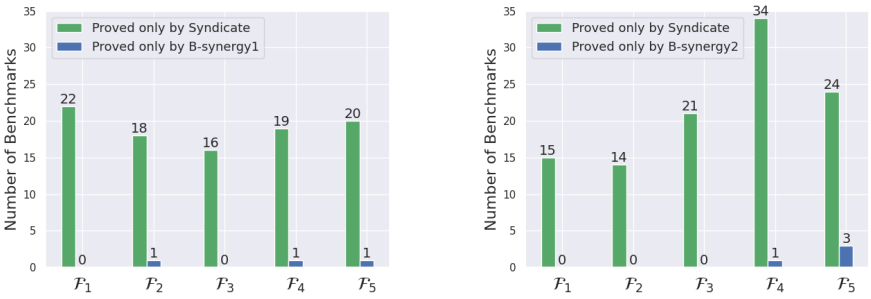
6.1 Efficacy of the Bidirectional Feedback

As discussed in Section 1, existing ranking-function-based termination analyzers fall into three categories: (1) those that search for ranking functions and invariants independently [10,17], (2) those with limited unidirectional feedback between the two searches [24,9,7,27], and (3) those that jointly synthesize both through a single monolithic query [3,22,23]. We show that the bidirectional feedback in SYNDICATE, where the invariant guides the ranking function and vice versa, proves more benchmarks in less time than either unidirectional or monolithic strategies, with consistent gains across all ranking-function templates.



(a) B-synergy1 and B-synergy2 (b) B-combined

Fig. 6: Percentage of benchmarks proved over time. (a) uses 168 benchmarks with $\mathcal{F}_3$, (b) uses 144 due to loop restrictions.



(a) Syndicate vs B-synergy1 (b) Syndicate vs B-synergy2

Fig. 7: Benchmarks uniquely solved by Syndicate compared to the baselines.

To evaluate this, we implemented three baselines: **B-synergy1**, **B-synergy2**, and **B-combined**. B-synergy1 ignores counterexamples from invariant search when refining ranking functions, while B-synergy2 ignores counterexamples from ranking-function search when refining invariants. B-combined jointly searches for both components using a single monolithic query, matching the formulation in VeryMax [16]. All baselines use the same templates as Syndicate and differ only in how the two searches interact. B-synergy1 and B-synergy2 are instantiated with the five templates $\mathcal{F}_i$ from Section 6, whereas B-combined is evaluated only with $\mathcal{F}_1$ due to scalability limits. All tools start with zero initial traces, and Syndicate, B-synergy1, and B-synergy2 use $\mathtt{P}_{\boldsymbol{ref}} = \mathtt{P}_{\boldsymbol{iter}} = 10$.

In Fig. 6a, we show the percentage of benchmarks proved over time for Syndicate, B-synergy1, and B-synergy2 under template $\mathcal{F}_3$. Both baselines perform worse than Syndicate, demonstrating that feedback in *both* directions is essential for proving more benchmarks within the timeout. Similar trends are observed

	Configuration				Performance	
	Parallel	Dynamic	Bounded	$P_{ref}, P_{iter} \neq \infty$	# Proved	Avg. Time (s)
SYNDICATE-BEST	✓	✓	✓	✓	**151**	**15.37**
SYNDICATE-SEQ	×	✓	✓	✓	147	18.35
SYNDICATE-STATIC	✓	×	✓	✓	**151**	15.91
SYNDICATE-UNBOUNDED	✓	✓	×	✓	150	16.83
SYNDICATE-FULL	✓	✓	✓	×	148	18.54

Table 4: Configurations and performance of different SYNDICATE variants.

for other templates (Appendix C). Table 3 reports the total benchmarks proved by each tool across all templates. SYNDICATE achieves up to 29% improvement over B-synergy1 and 35% over B-synergy2. Since all configurations use identical templates, the performance gains arise solely from the synergistic exchange of guidance between the two searches. Figure 7 shows benchmarks uniquely solved by SYNDICATE, again highlighting its advantage. The few cases where baselines outperform SYNDICATE are due to non-determinism in candidate generation. Finally, Fig. 6b compares SYNDICATE with B-combined under $\mathcal{F}_1$. As B-combined supports only non-nested loops, we evaluate it on 144 benchmarks. It solves 31 benchmarks quickly, since its single-query formulation requires fewer iterations, but times out on more complex ones due to the enlarged search space. This shows that the monolithic strategy does not scale with benchmark complexity.

```
int t;
if (b >= 1) {
   t = 1;
} else {
   t = -1;
}
while (x <= n) {
   if (b >= 1) {
      x = x + t;
   } else {
      x = x - t;
   }
}
```

Fig. 8: Failure Example

Failure Analysis of SYNDICATE. We also analyze the benchmarks that SYNDICATE-BEST fails to prove terminating within the timeout. Out of all programs, 17 remain unproved. For 9 of them, no ranking functions or invariants exist within the chosen templates, as confirmed by SYNDICATE-FULL ($P_{ref} = P_{iter} = \infty$), which terminates and proves the absence of a termination argument. Among the remaining 8, `refine` fails to find invariants ruling out unreachable states in 4 cases, causing SYNDICATE to loop over invariant generation, while in the other 4, it keeps generating new ranking functions until timeout. Fig. 8 shows a program whose termination SYNDICATE-BEST cannot prove using the templates in our implementation. The example admits no ranking function within these templates, whereas tools such as Aprove and Ultimate succeed by using algebraic bounds and multiple region-specific ranking functions, respectively. Proving termination with SYNDICATE would require richer templates, such as conditional or polynomial ones.

Variations within SYNDICATE. SYNDICATE supports several variants built on the Bidirectional Decomposition Search (BDS) concept, each exploring different trade-offs for distinct benchmark classes. All variants share the same synergistic search strategy, while features such as templates, parallelism, dynamic

analysis, and completeness are varied to isolate their impact. Table 4 summarizes these configurations and their performance. SYNDICATE-SEQ uses a single template $\mathcal{F}_3$ without parallelism, SYNDICATE-STATIC performs a fully static analysis with zero initial traces, SYNDICATE-UNBOUNDED removes coefficient bounds, yielding infinite search spaces, and SYNDICATE-FULL sets $\mathtt{P}_{ref} = \mathtt{P}_{iter} = \infty$, making the procedure complete for single-loop programs. SYNDICATE-BEST, used in our main experiments, runs five templates $(\mathcal{F}_1, \ldots, \mathcal{F}_5)$ in parallel, initializes with 100 traces, bounds coefficients by 10,000, and sets $\mathtt{P}_{ref} = \mathtt{P}_{iter} = 10$. Table 4 shows that while SYNDICATE-BEST achieves the highest precision and efficiency, the other variants also perform strongly (and outperform existing analyzers), demonstrating the effectiveness of the BDS strategy.

6.2 Comparison with State-of-the-Art Termination Analysis Tools

This section compares SYNDICATE with several state-of-the-art termination analyzers. Unlike SYNDICATE, which relies on a single ranking function per loop and invariants synthesized using standard SMT solving, existing tools employ diverse and often sophisticated techniques such as neural networks, multiple ranking functions for different loop components, reduction orders, and solver-specific optimizations. Despite its simpler design, SYNDICATE consistently performs on par with, or better than, these advanced systems, proving a broader range of benchmarks and reducing average runtime by 16%–71%.

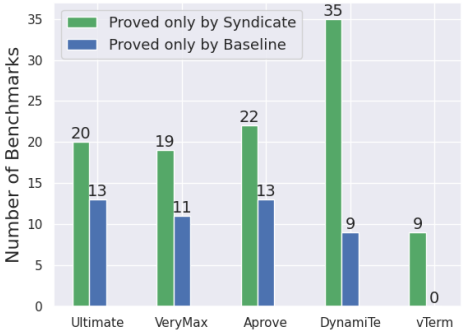
Tools compared. We compare SYNDICATE against state-of-the-art termination analyzers. ULTIMATE [14] constructs ranking functions for sub-programs, then checks whether the current set of sub-programs captures the behavior of the entire loop. VeryMax [3] combines multiple ranking functions, searching for preconditions under which each is valid. Its design resembles our B-combined baseline but includes SMT-level optimizations to ensure solver scalability. AProVE [11] integrates multiple proof techniques, including ranking functions, reduction orders, and dependency pairs. DynamiTe [17] combines dynamic and static reasoning and relies on the invariant generator DIG [19]. νTerm [10] uses neural networks to learn ranking functions from traces while enumerating invariants from a fixed library. A detailed summary of these tools is provided in Table 5 (Appendix F). In contrast, SYNDICATE achieves superior performance by using bidirectional feedback between the invariant and ranking-function searches.

Parallelism. ULTIMATE, VeryMax, and AProVE have previously won the Termination Competition [12], and AProVE has also won the Software Verification Competition [2]. All tools employ parallel implementations: AProVE and VeryMax are fully parallelized, ULTIMATE and DynamiTe parallelize ranking-function validation, and both νTerm and SYNDICATE parallelize trace generation. In our experiments, all tools were allowed to use all available CPU cores.

Results. Table 9a summarizes the number of benchmarks each tool proves and the average runtime. SYNDICATE proves the largest number of benchmarks in the least average time, achieving efficiency gains of 35.1% over ULTIMATE, 16.0% over VeryMax, 31.1% over AProVE, and 71.0% over DynamiTe. Notably, SYNDICATE outperforms AProVE and VeryMax despite their full parallelization

	# Proved	Avg. Time (s)
SYNDICATE-BEST	**151**	**15.37**
ULTIMATE	144	23.69
VeryMax	143	18.30
AProVE	142	22.31
DynamiTe	126	53.04
νTerm	96*	33.13*

(a) Benchmarks proved and average runtime.



(b) Pairwise comparison with baselines.

Fig. 9: Comparison of SYNDICATE with state-of-the-art termination analyzers. (a) Benchmarks (out of 168) proved terminating and average runtime per benchmark. Other SYNDICATE variants (Table 4) also prove more benchmarks within the timeout. (∗) Results for νTerm are based on 123 benchmarks. (b) Pairwise comparison of benchmarks uniquely proved by SYNDICATE and by each baseline.

and specialized SMT optimizations. νTerm supports only 123 benchmarks, as it does not handle sequential loops or random-number calls. Using identical templates ($\mathcal{F}_4$) and νTerm's best configuration (1000 traces), SYNDICATE proves 105 benchmarks with an average runtime of 24.86s, compared to νTerm's 96 benchmarks and 33.13s, yielding a 25% runtime improvement. From Tables 4 and 9a, we observe that all variants of SYNDICATE outperform existing techniques in both coverage and runtime. Even the sequential variant (SYNDICATE-SEQ) using a single template ($\mathcal{F}_3$) surpasses fully parallelized baselines. Fig. 9b highlights the number of benchmarks uniquely proved by SYNDICATE-BEST versus each baseline, demonstrating the advantage of SYNDICATE 's bidirectional decompositional search approach over traditional termination analyzers.

6.3 Case Studies: Benefits of Bidirectional Decompositional Search

To further illustrate the efficacy of SYNDICATE, we discuss two non-trivial benchmarks that existing tools fail to solve within the timeout, while SYNDICATE succeeds in 1.8s and 5.4s, respectively. We compare qualitatively with DynamiTe and νTerm, which most closely resemble components of SYNDICATE.

To prove the termination of Program 10a, it suffices to show that x decreases in successive iterations. Algebraic rewriting reveals that both x and y decrease by the initial value of $x-y$. One needs an invariant at least as strong as $x-y>0$ to show that x always reduces. Although the termination argument lies within their search space, AProVE and ULTIMATE fail to prove this benchmark. SYNDICATE succeeds in 1.8s through its synergistic search, yielding $f^* \equiv \max(x+1,0)$

```
int x = y + 42;
while (x >= 0) {
    y = 2 * y - x;
    x = (y + x) / 2;
}
```

(a) Not proved by AProVE or Ultimate

```
int a = 0, b = 0;
while (a*b<=n || a*b<=m) {
    a = a + 1;
    b = b + 1;
}
```

(b) Not proved by any existing tool

Fig. 10: Benchmarks for Case Studies

and $\mathsf{I}^* \equiv x - y \geq 1$. DynamiTe, which also uses traces to propose candidate ranking functions, is five times slower (11.9s) because its invariant generation, handled by DIG, is not guided by the ranking-function search. Also, the traces DynamiTe uses to generate ranking functions are not guided by the invariant search. Stronger invariants such as $x - y \geq 42$ can also prove f^* but require longer search times due to the unguided search for invariants. νTerm proves termination in 2.56s, but relies on non-generalizable, hard-coded invariants.

The second program in Fig. 10 involves a non-linear loop guard that none of the other tools can handle. Syndicate proves termination in 5.4s by synthesizing a lexicographic ranking function $\langle \max(n{-}2m{+}a{-}4b{+}1, 0), \max(m{+}a{-}2b{+}6, 0)\rangle$ and invariant $a \geq b$. DynamiTe fails because DIG does not infer invariants useful for validating ranking functions within the timeout. νTerm also fails despite the existence of a valid ranking function in its template, as it cannot guess the necessary invariants. These examples show that guiding both the invariant and ranking-function searches, as in Syndicate increases the number of benchmarks proved terminating and significantly reduces analysis time.

7 Related Works

Combined Search. Numerous techniques have been designed to reason about program termination [10,24,9,26,17,13,20,6,7,16,3,28,22,23]. Some of these techniques [23,22,16,3,28] directly search for both the ranking function and over-approximation of the reachable states together with one logical formula, which can be prohibitively expensive to solve, since the size of the search space would be the product of the sizes of the individual ranking function and invariant search spaces. Existing techniques based on SMT queries [23,22,16,3] introduce optimizations to efficiently solve each monolithic SMT query, but provide no theoretical guarantees on how efficiently they can navigate the combined search space to find a valid combination of ranking function and invariants. Instead, Syndicate separates the ranking function and invariant searches while exchanging sufficient bidirectional feedback to efficiently navigate through the space of possible ranking functions and invariants, as described in §4.3.

Unidirectional Feedback. Some termination tools [17,9,7,24,27,15] invoke external invariant generators or safety checkers that implicitly synthesize invariants after producing candidate ranking functions. These methods typically provide

no feedback to the ranking-function search beyond the final invariants. As a result, the invariant search cannot guide ranking-function synthesis, and each call to the external engine must restart its search since prior invariant states are not reused. In contrast, SYNDICATE uses its `refine` procedure to guide both subsequent `refine` iterations and future ranking-function generation. As shown in § 6.1, allowing invariants to guide ranking-function synthesis significantly increases the number of benchmarks proved within the timeout compared to unidirectional feedback. Similar to how SYNDICATE exploits synergy between invariants of multiple loops and the ranking function of one loop, [8] shares information about preconditions needed for one function's termination in an inter-procedural analysis. While [8] shares information that affects the initial states of other code, the use of annotated programs to define this synergy within SYNDICATE allows it to have efficiency and completeness guarantees.

Formal Guarantees. Some termination analysis techniques offer completeness guarantees but apply only to restricted program classes [13,20,5], such as those admitting linear ranking functions. [28] provides relative completeness for polyhedral ranking functions by statically computing a finite set of candidates for each loop. In contrast, SYNDICATE offers relative completeness for broader classes of non-linear ranking functions, including those with logical operators. [23] and [22] also establish relative completeness for general ranking functions, but neither provides completeness and efficiency guarantees simultaneously. [22] attains scalability empirically by leveraging information from a parallel non-termination analysis. SYNDICATE, in contrast, focuses solely on termination, maintaining provable efficiency while ensuring relative completeness. It also allows users to tune efficiency for specific use cases without sacrificing completeness guarantees.

8 Conclusion

We presented SYNDICATE, a general framework for automated termination analysis based on BDS. SYNDICATE supports diverse templates, is efficient, relatively complete, and scales to complex programs. By changing only the search strategy using BDS and without using mechanisms like reduction orders or SMT-level optimizations employed by existing solvers, SYNDICATE proves more benchmarks, including some that no other tool can, in less average time, demonstrating the efficacy of BDS. Currently, SYNDICATE is limited to the syntax defined in § 3. Extending the theory to support arbitrary function calls, memory allocation, threads, and higher-order types remains an important direction for future work.

9 Acknowledgements

This work was supported in part by NSF Grants No. CCF-2238079, CCF-2316233, CNS-2148583, and an Amazon research award. This work used JetStream2 at Indiana University through allocation CIS 240285 from the NSF Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program.

10 Data-Availability Statement

An artifact containing the implementation of Syndicate, along with scripts and documentation necessary to reproduce the experimental results in this paper, is available at `https://doi.org/10.5281/zenodo.18195083`. The latest development version is maintained at `https://github.com/uiuc-focal-lab/Syndicate`.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. pp. 113–132 (10 2007). `https://doi.org/10.1007/978-3-540-92188-2_5`
2. Beyer, D.: Advances in automatic software verification: Sv-comp 2020. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 347–367. Springer International Publishing, Cham (2020)
3. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205. p. 99–117. Springer-Verlag, Berlin, Heidelberg (2017). `https://doi.org/10.1007/978-3-662-54577-5_6`, `https://doi.org/10.1007/978-3-662-54577-5_6`
4. Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
5. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification. pp. 491–504. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In: Cousot, R. (ed.) Verification, Model Checking, and Abstract Interpretation. pp. 113–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
7. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 413–429. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. p. 53–64. ASE '15, IEEE Press (2015). `https://doi.org/10.1109/ASE.2015.10`, `https://doi.org/10.1109/ASE.2015.10`
9. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 415–426. PLDI '06, Association for Computing Machinery, New York, NY, USA (2006). `https://doi.org/10.1145/1133981.1134029`, `https://doi.org/10.1145/1133981.1134029`
10. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 633–645. ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA (2022). `https://doi.org/10.1145/3540250.3549120`, `https://doi.org/10.1145/3540250.3549120`

11. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with aprove. J. Autom. Reason. **58**(1), 3–31 (jan 2017). https://doi.org/10.1007/s10817-016-9388-y, https://doi.org/10.1007/s10817-016-9388-y
12. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 156–166. Springer International Publishing, Cham (2019)
13. Gonnord, L., Monniaux, D., Radanne, G.: Synthesis of ranking functions using extremal counterexamples. SIGPLAN Not. **50**(6), 608–618 (jun 2015). https://doi.org/10.1145/2813885.2737976, https://doi.org/10.1145/2813885.2737976
14. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs (05 2014). https://doi.org/10.1007/978-3-319-08867-9_53
15. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Leveraging llms for program verification. In: Formal Methods in Computer-Aided Design (FMCAD) (October 2024)
16. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination of imperative programs using max-smt. In: 2013 Formal Methods in Computer-Aided Design. pp. 218–225 (2013). https://doi.org/10.1109/FMCAD.2013.6679413
17. Le, T.C., Antonopoulos, T., Fathololumi, P., Koskinen, E., Nguyen, T.: Dynamite: Dynamic termination and non-termination proofs. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428257, https://doi.org/10.1145/3428257
18. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. vol. 45, pp. 211–222 (01 2010). https://doi.org/10.1145/1706299.1706326
19. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Dig: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. **23**(4) (sep 2014). https://doi.org/10.1145/2556782, https://doi.org/10.1145/2556782
20. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 239–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
21. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. Autom. Reason. **58**(1), 33–65 (Jan 2017). https://doi.org/10.1007/s10817-016-9389-x, https://doi.org/10.1007/s10817-016-9389-x
22. Unno, H., Terauchi, T., Gu, Y., Koskinen, E.: Modular primal-dual fixpoint logic solving for temporal verification. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). https://doi.org/10.1145/3571265, https://doi.org/10.1145/3571265
23. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 742–766. Springer International Publishing, Cham (2021)
24. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Chechik, M., Raskin, J.F. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 54–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

25. Welp, T., Kuehlmann, A.: Property directed invariant refinement for program verification. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1–6 (2014). https://doi.org/10.7873/DATE.2014.127
26. Xie, X., Chen, B., Zou, L., Lin, S.W., Liu, Y., Li, X.: Loopster: Static loop termination analysis. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. p. 84–94. ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3106237.3106260, https://doi.org/10.1145/3106237.3106260
27. Xu, R., Chen, J., He, F.: Data-driven loop bound learning for termination analysis. In: Proceedings of the 44th International Conference on Software Engineering. p. 499–510. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510220, https://doi.org/10.1145/3510003.3510220
28. Zhu, S., Kincaid, Z.: Breaking the mold: Nonlinear ranking function synthesis without templates. In: Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I. p. 431–452. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65627-9_21, https://doi.org/10.1007/978-3-031-65627-9_21

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.