# Accurate Inference of Termination Conditions

Biting Huang
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hbt23@mails.tsinghua.edu.cn

Zhilei Han
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hzl21@mails.tsinghua.edu.cn

Fei He*
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@tsinghua.edu.cn

## ABSTRACT

We present the first approach to infer termination conditions that are accurate in the sense of precisely characterizing terminating states. It builds on a simple but effective framework where non-terminating states are iteratively identified and removed, and a termination prover is employed to validate the current condition. We instantiate the framework with data-driven provers and design a multi-way data sharing mechanism to enhance their interaction. Our proofs show that this method is correct, accurate, terminating, and relatively complete. Additionally, we introduce generalization techniques for recurrent sets to accelerate the iteration process. Evaluation on a benchmark of programs from the literature shows that our implementation significantly outperforms the state-of-the-art tool Acabar, producing much more accurate termination conditions, with the proposed techniques playing a crucial role in speeding up the convergence of the process.

## CCS CONCEPTS

• **Theory of computation → Logic and verification**.

## KEYWORDS

program termination, termination condition, recurrent set, black-box learning

**ACM Reference Format:**
Biting Huang, Zhilei Han, and Fei He. 2026. Accurate Inference of Termination Conditions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764568

## 1 INTRODUCTION

Termination is an essential property to establish total correctness of programs. As a classical problem in formal verification, it has been extensively studied in the past, and considerable termination analysis tools [3, 23, 28, 31] have emerged in the last decade. On the other hand, since temination is generally undecidable, proving non-termination, the dual property to termination, is another active research topic. A wide range of non-termination analysis methods have been proposed recently, including white-box analyses [9, 33, 40] and "guess-and-check" techniques [18, 26, 35]. To provide witness for non-termination, most of them employ the concept of recurrent set [10, 25] as the non-termination argument.

Termination and non-termination analyses both answer a "yes-or-no" question about program behavior. However, most programs are neither always terminating nor always non-terminating – they may terminate on some inputs but not on others. Traditional (non-)termination analysis typically provides a binary answer, which is insufficient for such cases where programs exhibit mixed behavior. This motivates the problem of conditional termination, which aims to more precisely characterize the conditions under which a program terminates.

Unlike termination and non-termination analyses, studies focusing on generating termination conditions remain limited. Among these, Cook et al. [12] introduce potential ranking functions and generate termination conditions by calculating preconditions that promote these functions to valid arguments. Another work [5] infers termination condition by over-approximating non-terminating states, while in [20] a condition is obtained by expanding an under-approximation of terminating states incrementally.

However, existing methods do not guarantee the precision of generated termination conditions and impose restrictions on programs (e.g. linear affine loops), limiting their applicability. Notably, *false* is a trivial but valid termination condition for any program, but its extreme imprecision renders it useless. Therefore, it is necessary to derive termination conditions with more accuracy, encompassing terminating states as much as possible.

In this paper, we introduce the first framework for inferring termination conditions that are accurate in the sense of precisely characterizing terminating states. This approach leverages existing (non-)termination provers to iteratively identify non-terminating states, remove them from the program, and verify termination of the remaining states. Once termination is proven, the termination condition is derived from all previously identified non-termination arguments. The framework is prover-agnostic and highly versatile. We instantiate it with data-driven provers and design a multi-way data sharing mechanism to enhance efficiency. To further reduce iterations, we introduce a generalization framework for recurrent sets, expanding the identified non-terminating states.

*Fei He is the corresponding author.


*ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764568

Based on this, we establish the correctness of our approach. First, we prove its soundness and accurateness, ensuring that the derived predicate exactly characterizes all terminating states. Second, we show that the method is guaranteed to terminate, provided that the accurate condition can be expressed as a boolean combination of certain atomic predicates, the non-termination prover is relatively complete, and the recurrent set identified in each iteration satisfies a bound condition (fully described in Section 4.4). Furthermore, if the termination prover is also complete under these conditions, our approach is guaranteed to return the accurate termination condition. To the best of our knowledge, no existing technique offers such theoretical guarantees on both precision and convergence.

We have implemented a prototype tool CondTerm based on the data-driven termination prover ddlTerm [55] and non-termination prover RSLearn [26]. Using the benchmarks from [55] and [26], we show that our approach can infer accurate termination conditions for both terminating and non-terminating programs. Additionally, we also compare with Acabar [20], the latest tool for generating termination conditions. Experimental results indicate that we can handle a significantly larger set of programs than Acabar and produce more accurate termination conditions.

Overall, the main contributions of this paper are as follows:

- We propose the first framework for inferring termination conditions that are accurate in the sense of precisely characterizing terminating states, and introduce a multi-way data sharing mechanism to enhance efficiency.
- We theoretically establish the correctness of the approach, specifically ensuring its soundness, accurateness, termination, and relative completeness.
- We introduce a generalization framework for recurrent sets and provide guarantees for its correctness.
- We implement our framework and demonstrate that it outperforms existing tools in both the number of solved cases and the precision of the generated conditions.

The remainder of the paper is organized as follows: Section 2 introduces basic concepts and background knowledge. Section 3 provides an overview of our approach through an illustrative example. Section 4 presents the framework for inferring accurate termination conditions.Section 5 details the technique to generalize recurrent set obtained from non-termination prover. Section 6 reports evaluation results of CondTerm. Section 7 reviews related work and Section 8 concludes the paper.

## 2 PRELIMINARIES

### 2.1 Basic Concepts

In this paper, we denote by $\vec{x}$ the set of program variables. A *state* $s$ is a valuation of variables, while a *transition* $(s, s')$ represents an execution from state $s$ to its successor $s'$. Transitions are described by *transition formula*, a formula over $\vec{x} \cup \vec{x}'$, where primed variables $\vec{x}'$ represent values after the transition. The combined valuation $s$ and $s'$ is given by $\sigma_{s,s'}$ where $\sigma_{s,s'}(v) = s(v)$ and $\sigma_{s,s'}(v') = s'(v)$. For a formula $\phi$, we say state $s \models \phi$ if the valuation of $s$ satisfies $\phi$. Similarly, for a transition formula $f$, we say transition $(s, s') \models f$ if the combined valuation of $s$ and $s'$ satisfies $f$.

Several existing techniques (e.g., [30]) can decompose programs into loops. This allows us to derive analysis results for programs by analyzing each loop individually.

*Definition 2.1 (Loop).* A *loop* $L$ is a triple $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, where $\mathcal{T}_{stem}$ and $\mathcal{T}_{loop}$ are transition formulas representing all transitions from the initial states to the loop entry and within the loop body respectively, and $\mathcal{G}_{loop}$ defines the loop's entry condition.

In analyzing a loop, we identify two types of states: the program's *initial states*, and *loop-head states*, which are states at the loop entry and are reachable from an initial state. A *trace* of a loop is represented as a sequence of loop-head states $s_1, s_2, \ldots$, such that:

(1) $(s_i, s_{i+1}) \models \mathcal{T}_{loop}$ for all $i \geq 1$,
(2) if the trace is finite with length $n$, then $s_i \models \mathcal{G}_{loop}$ for $1 \leq i < n$ and $s_n \not\models \mathcal{G}_{loop}$; if the trace is infinite, then $s_i \models \mathcal{G}_{loop}$ for all $i \geq 1$.

Throughout this paper, it is assumed that $\mathcal{T}_{loop}$ is deterministic, i.e., $\forall x, x', x'', \mathcal{T}_{loop}(x, x') \wedge \mathcal{T}_{loop}(x, x'') \rightarrow x' = x''$. Each loop-head state then induces a *trace* of the loop. A loop-head state $s$ is said to be *terminating* (resp. *non-terminating*) if the trace starting from $s$ is finite (resp.infinite). Furthermore, a loop $L$ is said to be *terminating* if each loop-head state of $L$ is terminating. Conversely, a loop is *non-terminating* if it has at least one non-terminating loop-head state, corresponding to an infinite trace.

For a loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ and a predicate $\varphi$, the *instrumentation* of $L$ with $\varphi$ is $L^{\varphi} = (\mathcal{T}^{\varphi}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, where $\mathcal{T}^{\varphi}_{stem}$ is given by $\forall \vec{x}, \vec{x}'. \mathcal{T}^{\varphi}_{stem}(\vec{x}, \vec{x}') \Leftrightarrow \mathcal{T}_{stem}(\vec{x}, \vec{x}') \wedge \varphi(\vec{x}')$. This is equivalent to inserting the statement "`assume (`$\varphi$`)`" immediately before the loop entry, ensuring that $\varphi$ holds when $L$ is first entered.

*Definition 2.2 (Termination Condition).* A formula $\varphi$ is called a *termination condition* of a loop $L$ if $L^{\varphi}$ is terminating.

Termination conditions characterize conditions under which the loop is guaranteed to be terminating. Moreover, we define the accurate termination condition, meaning that the program will terminate if and only if the condition holds.

*Definition 2.3 (Accurate Termination Condition).* A termination condition $\varphi$ is *accurate* if it precisely characterizes the set of all terminating loop-head states. In other words, any loop-head state $s \not\models \varphi$ must be non-terminating.

For brevity, in the remainder of this paper, we will often abbreviate "terminating/non-terminating loop-head states" as simply "terminating/non-terminating states".

### 2.2 Recurrent Set

To prove non-termination, the standard approach is to identify a *recurrent set*, a set of states from which execution can repeat indefinitely without getting out of the set. The existence of such a set guarantees an infinite trace, establishing non-termination.

*Definition 2.4 (Recurrent Set).* A predicate $\mathbf{R}$ is a *recurrent set* of a deterministic loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ if it satisfies the recurrent set constraints $\mathcal{RSC}(\mathbf{R}) := \mathcal{RC}(\mathbf{R}) \wedge \mathcal{GC}(\mathbf{R}) \wedge \mathcal{IC}(\mathbf{R})$, where:

- $\mathcal{RC}(\mathbf{R}) \Leftrightarrow \exists \vec{x}, \vec{x}'. \mathcal{T}_{stem}(\vec{x}, \vec{x}') \wedge \mathbf{R}(\vec{x}')$ *(reachability)*
- $\mathcal{GC}(\mathbf{R}) \Leftrightarrow \forall \vec{x}. \mathbf{R}(\vec{x}) \rightarrow \mathcal{G}_{loop}(\vec{x})$ *(guard)*

- $\mathcal{IC}(\mathbf{R}) \Leftrightarrow \forall \vec{x}, \vec{x}'.\ \mathbf{R}(\vec{x}) \wedge \mathcal{T}_{loop}(\vec{x}, \vec{x}') \rightarrow \mathbf{R}(\vec{x}')$ *(inductiveness)*

In the definition, $\mathcal{RC}(\mathbf{R})$ ensures that $\mathbf{R}$ is reachable from the beginning of the loop; $\mathcal{GC}(\mathbf{R})$ guarantees that each state in $\mathbf{R}$ satisfies the loop guard, thus does not terminate; $\mathcal{IC}(\mathbf{R})$ demonstrates the inductiveness of $\mathbf{R}$, showing that states in $\mathbf{R}$ remain within the set after each loop iteration. Therefore, once the program execution reaches $\mathbf{R}$, it will continue to execute indefinitely within $\mathbf{R}$, ensuring the existence of a reachable infinite trace. For a predicate $\mathbf{H}$, if $\mathcal{RSC}(\mathbf{H})$ is valid, we say $\mathbf{H}$ defines a *valid recurrent set* of $L$.

## 2.3 Black-box Learning

Black-box learning is a powerful approach for program analysis that consists of two components: a learner and a teacher. It treats the program as a black box, extracting samples without examining its internal structure. The learner uses these samples to infer a target hypothesis, while the teacher validates it. If valid, a desired proof is established. Otherwise, the teacher provides counterexamples for further learning iterations. This approach decouples the learning process from the program's semantics, enabling it to handle more complex cases, such as non-linear assignments.

Program analysis based on the black-box framework typically relies on decision tree learning, a well-studied machine learning technique with many established algorithms like ID3 [48], C4.5 [49] and C5.0. Decision tree learning has already been applied in both termination analysis [55] and non-termination analysis [26], leveraging its structured approach to classify program behaviors and facilitate the learning of (non-)termination arguments.

Specifically, an *attribute* is a numerical expression over $\vec{x}$, with different methods focusing on different attributes. For example, the octagonal domain includes attributes $\{\pm x \pm y | x, y \in \vec{x}\}$. Given a finite set of attributes $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, an *atomic predicate* takes the form $a \leq c$, where $a \in \mathcal{A}$ and $c$ is a constant. A *decision tree* is a boolean combination of atomic predicates. In decision tree learning, the learner selects the most suitable atomic predicate $a \leq c$ based on information gain, splitting the state space into two regions: one where $a \leq c$ holds and one where it does not. This process continues iteratively, refining the state space and producing a structured decision tree that best fits the samples.

A widely-used decision tree learning framework for verification is the ICE framework [21], which classifies samples into three categories: positive, negative and implicative. Positive samples $S^+$ (resp. negative samples $S^-$) represent program states that must be included in (resp. excluded from) the target predicate. An implicative sample $(s_1, s_2) \in S^{\rightarrow}$ signifies that if $s_1$ satisfies the target predicate, then so does $s_2$. The ICE framework learns a decision tree that satisfies all conditions imposed by the sample set $S = (S^+, S^-, S^{\rightarrow})$.

# 3 AN ILLUSTRATIVE EXAMPLE

In this section, we illustrate the core idea of our approach through a simple example. Consider the loop $\mathbf{L}$ in Figure 1, where $t$ and $w$ are integer variables of the program.

By analyzing the behavior of the loop, we can observe that when $w \geq 0$, if $-w \leq t \leq w$, $t$ is set to 0 in the loop body and subsequently exits the loop. If $t < -w$ or $t > w$, the loop body increments the absolute value of $t$ by 1 and flips its sign, causing $t$ to move further away from 0 with each iteration, resulting in infinite execution. When $w < 0$ and $t \neq 0$, the loop terminates if and only if $t < -w$ and $t = 1$, i.e. $w <= -2 \wedge t = 1$. Otherwise, $t$ will move away from 0 and the loop executes infinitely. Additionally, when $t = 0$, the loop always terminates as the loop guard condition fails.

```
1 while (t != 0) {
2   if (t < -w) {
3     t := t - 1;
4     t := t * (-1);
5   }
6   else if (t > w) {
7     t := t + 1;
8     t := t * (-1);
9   }
10  else {
11      t := 0;
12  }
13 }
```

**Figure 1: An illustrative example.**

Notably, many predicates can serve as termination conditions for this loop, but not all are useful. For example, the trivial predicate *false* technically qualifies but excludes all executions, providing no useful information about the loop's behavior. Our objective is to infer an accurate termination condition (Section 4).

We begin by guessing *true* as the initial conjecture for the accurate terminating condition, denoted as $\varphi_0$. To validate whether $\varphi_0$ is an accurate termination condition, we attempt to prove the termination of $\mathbf{L}^{\varphi_0}$, the instrumented version of $\mathbf{L}$ with $\varphi_0$. Since the statement `assume (`*true*`)` has no effect on the original loop, the termination proving of $\mathbf{L}^{\varphi_0}$ must fail.

Next, we turn to a non-termination prover to check if $\mathbf{L}^{\varphi_0}$ is non-terminating. Given that the instrumented loop is indeed non-terminating, we assume the non-termination prover returns a recurrent set $\mathbf{R}_1^o : w = 0 \wedge (t \leq -2 \vee 0 < t)$, which characterizes a set of non-terminating states. To expedite the iterative process, we apply a generalization technique to expand the returned recurrent set. For instance, $\mathbf{R}_1^o$ is generalized to $\mathbf{R}_1 : w = 0 \wedge (t \leq -1 \vee 0 < t)$. This generalization process (Section 5) is guaranteed to preserve the validity of the recurrent set, meaning the expanded states must also be non-terminating.

As all states identified by $\mathbf{R}_1$ are confirmed as non-terminating, the accurate termination condition must exclude these states. Consequently, we refine our conjecture $\varphi_0$ to $\varphi_1 := \varphi_0 \wedge \neg\mathbf{R}_1 \equiv w \neq 0 \vee (-1 < t \wedge t \leq 0)$, thereby excluding the non-terminating states identified by $\mathbf{R}_1$. The refined conjecture $\varphi_1$ is then used to instrument the loop $\mathbf{L}$, yielding $\mathbf{L}^{\varphi_1}$.

We now apply the same process to $\mathbf{L}^{\varphi_1}$. Once again, the termination proving for $\mathbf{L}^{\varphi_1}$ fails, while the non-termination proving yields a new recurrent set, which, after generalization, yields $\mathbf{R}_2$. Accordingly, we refine the conjecture further to $\varphi_2 := \varphi_1 \wedge \neg\mathbf{R}_2$, and obtain the new instrumented loop $\mathbf{L}^{\varphi_2}$. This process continues iteratively until the 9th refinement, resulting in the final conjecture: $\varphi_9 : (w \geq 0 \wedge -w \leq t \leq w) \vee t = 0 \vee (w \leq -2 \wedge t = 1)$. At this point, the instrumented loop $\mathbf{L}^{\varphi_9}$ successfully passes the termination proof.

Given the soundness and accurateness of our approach (established in Section 4.4), we can confidently conclude that $\varphi_9$ is an

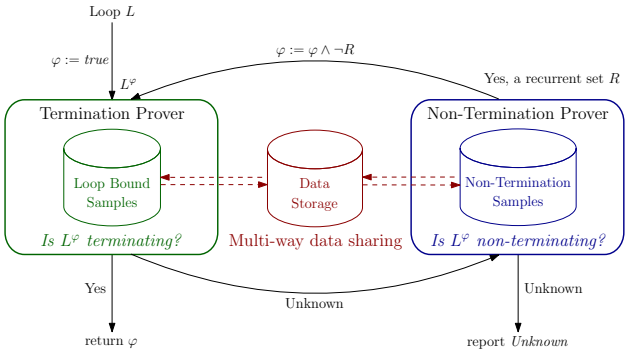


**Figure 2: The inference framework.**

accurate termination condition for the original loop **L**. Additionally, we have established the termination and relative completeness of our approach (in Section 4.4). This ensures that, provided the underlying termination and non-termination provers meet the specified requirements, the iterative procedure will always terminate in a finite number of steps and return an accurate termination condition for any given loop.

## 4 INFERENCE FRAMEWORK

Termination is a challenging property that requires all traces of a loop to be finite. For loops that do not satisfy this property, our goal is to infer an accurate termination condition that characterizes the maximal set of loop-head states that ensure finite loop traces. This section describes the process for inferring such conditions.

### 4.1 Overall Procedure

Figure 2 illustrates the overall procedure of our approach, with its pseudocode in algorithm 1. Given a loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, the goal is to infer the accurate termination condition $\varphi$ of the loop. If no such condition can be inferred, the procedure returns *Unknown*. Let $\varphi$ represent the current conjecture for the accurate termination condition. The procedure begins by initializing $\varphi$ to *true* (Line 1 in algorithm 1), and then iteratively refines $\varphi$.

In each iteration, the procedure updates the loop $L$ to $L^\varphi = (\mathcal{T}^\varphi_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ (Line 3). As explained earlier, $\mathcal{T}^\varphi_{stem}$ is defined as $\forall \vec{x}, \vec{x}', \mathcal{T}^\varphi_{stem}(\vec{x}, \vec{x}') \Leftrightarrow \mathcal{T}_{stem}(\vec{x}, \vec{x}') \wedge \varphi(\vec{x}')$, which is equivalent to inserting the statement "`assume` $(\varphi)$" before the loop entry. Thus, only executions where $\varphi$ holds upon the initial entry into $L$ are taken into account.

The instrumented loop $L^\varphi$ is then passed to a termination prover (Line 4). If termination is established, the procedure stops and returns $\varphi$ as the inferred accurate termination condition. Otherwise, a non-termination prover is invoked to determine whether $L^\varphi$ exhibits non-termination (Line 7). If successful, the prover returns a recurrent set **R** as the non-termination argument. Otherwise, neither termination nor non-termination of $L^\varphi$ can be proved, the procedure stops and returns *Unknown* (Line 9).

Each recurrent set represents a set of loop-head states that are confirmed to be non-terminating. Note that states in recurrent sets might not be reachable since the reachability constraint $\mathcal{RC}(\mathbf{R})$ in Definition 2.4 is existential, but these states can be safely excluded from further consideration in the accurate termination condition. This is achieved by refining $\varphi$ to $\varphi \wedge \neg\mathbf{R}$ (Line 10). The above process repeats until either an accurate termination condition is inferred or the algorithm reports *Unknown*.

**Algorithm 1:** Inference of Accurate Termination Conditions.

**input** : A loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$
**output**: An accurate termination condition $\varphi$ of $L$, or "*Unknown*" if such condition cannot be inferred.

```
1  φ := True
2  while True do
3      L^φ := (T^φ_stem, G_loop, T_loop)
4      rst := termination_prover (L^φ)
5      if rst is "Yes" then
6          return φ
7      (rst, R) := nontermination_prover (L^φ)
8      if rst is "Unknown" then
9          return "Unknown"
       // R := rs_generalization (R)
10     φ = φ ∧ ¬R
```

### 4.2 Termination and Non-Termination Provers

The versatile framework in algorithm 1 allows for the use of various termination provers and non-termination provers, a considerable number of which has been developed over the years and manifests varying abilities to handle different classes of programs.

However, since the (non-)termination provers in our framework are treated as black-boxes and take the same instrumented program as input, we prefer provers that are effective, fully automated and apply similar and coherent algorithms. In that regard, we adopt two newly proposed data-driven provers, which employ black-box learning as introduced in Section 2.3 and inherit all its benefits.

Recall that in such data-driven provers, the target proof argument is synthesized from a sample set, with samples classified into three categories: positive, negative, and implicative. Positive (resp. negative) samples indicate that the required argument must hold (resp. not hold) on these samples, while implicative samples represent the transition constraints that the argument needs to satisfy.

These samples are either statically obtained or dynamically observed from the program, though the exact data represented by a sample may varies slightly among provers. For a data-driven termination prover such as ddlTerm [55], the goal is to learn a correct loop bound from loop bound samples to serve as the program's termination argument. Therefore, each sample includes a program state and its corresponding loop bound. On the other hand, a data-driven non-termination prover (such as RSLearn [26]) aims to learn a valid recurrent set from a set of non-termination samples, which includes a program state as well as its termination behavior (i.e., whether it terminates or not).

Moreover, with regard to the validation of the inferred candidate argument, termination provers rely on a safety checker to verify the correctness of a candidate bound, which entails deriving a suitable loop invariant. In this context, the widespread adoption of data-driven approaches is exemplified by the ICE framework [21], which employs positive, negative, and implicative samples for invariant learning and has been applied in ddlTerm. On the other hand, for non-termination provers, the validation of a recurrent set is performed using an SMT solver, which checks whether the candidate satisfies the formal definition of a recurrent set.

## 4.3 Multi-way Data Sharing

In our framework, the termination and non-termination provers alternately analyze the program. The provers individually need to start verification from an empty sample set, though the samples generated for one prover are likely to be helpful for the other. Meanwhile, the provers is invoked multiple times across different iterations. Although each invocation targets a slightly modified version of the program, the terminating states and the loop body remain unchanged. This motivates us to introduce interaction and data sharing between these two provers as well as across different iterations, which can further improve the efficiency of the scheme.

In our framework, we design a multi-way data sharing shown as the red part in Figure 2. Samples found in the two provers can be stored and reused during the whole process. Specifically, the data sharing mechanism is illustrated in Figure 3.

As previously discussed, the termination prover and the non-termination prover together incorporate three data-driven components: loop bound learning, invariant learning and recurrent set learning, as shown in Figure 3. The samples corresponding to these three types of learning algorithms are as follows:

- Loop bound samples $\mathcal{H}$ consist of samples $(\vec{x}, r)$ where $\vec{x}$ represents a loop-head program state and $r$ is the corresponding loop bound.
- Invariant samples $S_{inv} = (S^+_{inv}, S^-_{inv}, S^{\rightarrow}_{inv})$ consist of positive, negative, and implicative samples for invariant learning. Each element in $S^+_{inv}$ or $S^-_{inv}$ represents a loop-head state, while each element in $S^{\rightarrow}_{inv}$ is a pair of two loop-head states.
- Recurrent set samples $S_{rs} = (S^+_{rs}, S^-_{rs}, S^{\rightarrow}_{rs})$ follow a similar structure to $S_{inv}$ but are used in the context of recurrent set learning, where positive samples $S^+_{rs}$ denote non-terminating loop-head states and $S^-_{rs}$ represent terminating ones.

Among these, two types of samples explicitly indicate termination: loop bound samples in $\mathcal{H}$, as a known loop bound guarantees termination, and negative recurrent set samples $S^-_{rs}$, which represent terminating states without an associated loop bound. Both categories are stored as terminating samples in our data storage. Additionally, we collect implicative samples from invariant learning and recurrent set learning, specifically $S^{\rightarrow}_{inv}$ and $S^{\rightarrow}_{rs}$, which aid in inferring additional terminating states.

During each learning phase of an iteration, the learner first attempts to retrieve and reuse terminating and implicative samples from the data storage, thereby avoiding the need for generating all samples. For the loop bound learner, if a reused terminating sample does not contain an loop bound, it is used as a test input for the program to obtain the corresponding loop bound by dynamic execution. As a result, the discovered samples can be shared across provers and iterations, therefore enhancing overall efficiency.

A simple data-sharing mechanism was previously introduced in [55] for data-driven termination analysis. However, that method was relatively limited, as it only facilitated information exchange between $\mathcal{H}$ and $(S^+_{inv}, S^-_{inv})$ while overlooking implicative samples. In contrast, our approach adopts a more comprehensive and effective sharing strategy by incorporating both terminating and implicative samples. These implicative samples are essential in the ICE learning framework, significantly improving its robustness and learnability [21].

## 4.4 Correctness

This section establishes the correctness properties of our approach. We begin by proving the soundness of algorithm 1, then address its accurateness, and finally discuss its termination and relative completeness.

The soundness of algorithm 1 follows directly from its construction: if the algorithm returns a predicate $\varphi$, this can only happen at Line 6, which indicates that the termination of $L^{\varphi}$ has been proven by the termination prover (Line 4 in algorithm 1). Thus, if the termination prover is sound, $\varphi$ must indeed be a valid termination condition for $L$. Here, soundness of the termination prover means that if it returns "*Yes*", then the loop passed to it must be terminating. This leads us to the following theorem:

Theorem 4.1 (Soundness). *Consider any loop represented as $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, if algorithm 1 returns a predicate $\varphi$, then $\varphi$ is guaranteed to be a termination condition of $L$, provided that the termination prover used in algorithm 1 is sound.*

Establishing the accurateness of algorithm 1 is more involved, requiring that both the termination and non-termination provers are sound. For a non-termination prover, soundness means that if it returns "*Yes*" along with a recurrent set, then all states within this set are indeed non-terminating.

Theorem 4.2 (Accurateness). *For any loop of the form $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, if algorithm 1 returns a predicate $\varphi$, then $\varphi$ is guaranteed to be an accurate termination condition for $L$, provided that the termination and non-termination provers used in algorithm 1 are sound.*

Proof. Suppose the algorithm returns $\varphi$ after $n$ complete iterations (i.e., Lines 3-10 in algorithm 1), with recurrent sets inferred in each iteration being $\mathbf{R}_1, \mathbf{R}_2, \cdots, \mathbf{R}_n$, respectively. According to algorithm 1, we have $\varphi := \neg\mathbf{R}_1 \wedge \neg\mathbf{R}_2 \wedge \cdots \wedge \neg\mathbf{R}_n \equiv \neg(\mathbf{R}_1 \vee \mathbf{R}_2 \vee \cdots \vee \mathbf{R}_n)$. By Theorem 4.1, $\varphi$ is a valid termination condition.

Now, assume for contradiction that $\varphi$ is not accurate enough. By Definition 2.3, there must exist a loop-head state $s \not\models \varphi$ such that $s$ is terminating. Given $s \not\models \varphi$, it follows that $s$ must belong to one of the recurrent sets $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_n$. However, this would contradict the soundness of the non-termination prover, which ensures that all states in each recurrent set $\mathbf{R}_i$ are non-terminating. Therefore, $\varphi$ must indeed be an accurate termination condition. □

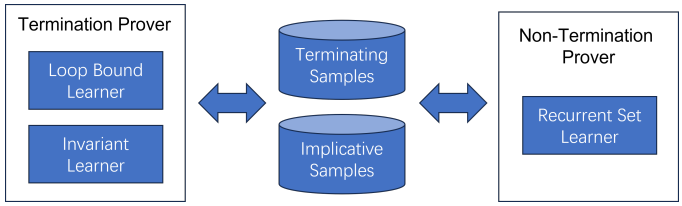


**Figure 3: Illustration of multi-way data sharing.**

The two theorems above establish the functional correctness of algorithm 1. We now proceed to demonstrate termination of the algorithm itself, i.e., after a finite number of iterations, the algorithm will always terminate by either returning an accurate termination condition or reporting *Unknown*. This property relies on the properties of the non-termination prover. Specifically, a non-termination prover is deemed *relatively complete* if it satisfies the following condition: given a set of attributes $\mathcal{A} = \{a_1, \cdots, a_n\}$, for any loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, the non-termination prover is guaranteed to terminate and return a recurrent set as long as $L$ has a recurrent set expressible as a boolean combination of predicates of the form $a \leq c$, where $a \in \mathcal{A}$ and $c$ is an integer. Furthermore, for any formula $\varphi$ expressed in this form, we define $maxc(\varphi)$ to be the maximum absolute value of all constants $c$ appearing in $\varphi$.

Theorem 4.3 (Termination). *Given a loop expressed as $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ and a set of attributes $\mathcal{A} = \{a_1, ..., a_n\}$, if the accurate termination condition $\varphi^*$ of $L$ can be expressed as a boolean combination of atomic predicates $a \leq c$ where $a \in \mathcal{A}$ and $c$ is any integer, then algorithm 1 is guaranteed to terminate, provided that the termination prover used in algorithm 1 is sound, the non-termination prover is both sound and relatively complete, and the recurrent set $R$ identified in each iteration by it satisfies $maxc(R) \leq maxc(\neg\varphi^*)$.*

Proof. In each iteration, the algorithm either terminates or infers a new recurrent set. Suppose the algorithm is at the $(i+1)$-th iteration, with the recurrent sets inferred in the preceding $i$ iterations being $\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_i$, so the current conjecture $\varphi$ is $\neg\mathbf{R}_1 \wedge \neg\mathbf{R}_2 \wedge \cdots \wedge \neg\mathbf{R}_i$. If the algorithm does not terminate at the current iteration, meaning that $L^\varphi$ remains non-terminating and a recurrent set is inferred. Since $\neg\varphi^*$, the negation of the accurate termination condition, characterizes all non-terminating states, there exists a loop-head state $s \models \neg\varphi^*$ that is reachable from the beginning of $L^\varphi$. Additionally, as $\neg\varphi^*$ represents all non-terminating states of $L$, it serves as a recurrent set for $L$ and, by extension, for $L^\varphi$. Therefore, there exists a recurrent set $R$ for $L^\varphi$ such that $maxc(R) \leq maxc(\neg\varphi^*)$.

Furthermore, for a finite attribute set $\mathcal{A}$ and an upper bound $m = maxc(\neg\varphi^*)$, the number of possible atomic predicates of the form $a \leq c$, where $a \in \mathcal{A}$ and $c$ is bounded by $m$, is finite. Consequently, the number of recurrent sets that can be expressed as boolean combinations of such predicates is also finite. Additionally, in each iteration, the algorithm excludes previously identified non-terminating states by the instrumentation of the loop. Thus, the non-termination prover will not infer the same recurrent set again. Finally, since the recurrent set $R$ identified in each iteration satisfies $maxc(R) \leq maxc(\neg\varphi^*)$, the algorithm is guaranteed to terminate after a finite number of iterations.

□

Notably, the termination prover ddlTerm is sound [55], and the non-termination prover RSLearn has been proven both sound and relatively complete [26]. What's more, according to Theorem 2 in [26], RSLearn can identify the recurrent set $R$ with the minimum $maxc(R)$, satisfying the premise of Theorem 4.3. Thus, instantiating our approach with ddlTerm and RSLearn ensures soundness, accurateness, and termination, establishing it as an effective framework for inferring accurate termination conditions for loops.

Furthermore, if the termination prover is complete, i.e., it can always successfully prove termination when the loop is indeed terminating, we can establish the relative completeness of our approach.

Theorem 4.4 (Relative Completeness). *Given a loop denoted by $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ and a set of attributes $\mathcal{A} = \{a_1, ..., a_n\}$, if the accurate termination condition $\varphi^*$ of $L$ can be expressed as a boolean combination of atomic predicates $a \leq c$ where $a \in \mathcal{A}$ and $c$ is any integer, then algorithm 1 is guaranteed to terminate with an accurate termination condition, provided that the termination and non-termination provers used in algorithm 1 are both sound and complete (or relatively complete), and the recurrent set $R$ identified in each iteration satisfies $maxc(R) \leq maxc(\neg\varphi^*)$.*

Proof. Theorem 4.2 and Theorem 4.3 ensure the algorithm's termination, with any resulting predicate $\varphi$ being a precise termination condition. Therefore, it suffices to prove that the algorithm is guaranteed to return a predicate, or equivalently, the algorithm will not terminate with *Unknown*.

Let $\varphi$ be the current conjecture of the condition. If $L^\varphi$ is passed to the non-termination prover, it must be non-terminating. Otherwise, the termination prover would have confirmed termination, leading to a halt with $\varphi$. As discussed in Theorem 4.3, $\neg\varphi^*$ serves as a recurrent set for $L^\varphi$. Since $\varphi^*$ can be expressed as a boolean combination of atomic predicates $a \leq c$ where $a \in \mathcal{A}$ and $c$ is an integer (i.e., expressible by $\mathcal{A}$), $\neg\varphi^*$ is expressible by $\mathcal{A}$ as well. Thus, $L^\varphi$ has at least one recurrent set expressible by $\mathcal{A}$. Given the relative completeness of the non-termination prover, it is guaranteed to return a recurrent set. Therefore, the algorithm will never terminate with *Unknown*, proving its relative completeness.

□

## 5 RECURRENT SET GENERALIZATION

Since non-termination verification is an existential problem, meaning that identifying a single state from which the program does not

**Algorithm 2:** Recurrent Set Generalization.

**input** : A DNF formula $\mathbf{R} : r_1 \vee r_2 \vee \cdots \vee r_n$
**output**: Generalized DNF formula $\mathbf{R}'$

1 **for** $i = 1$ **to** $n$ **do**
2 $\quad r_i \leftarrow$ `predicate_elimination` $(r_i, \mathbf{R})$
3 $\quad r_i' \leftarrow$ `constant_generalization` $(r_i, \mathbf{R})$
4 $\quad \mathbf{R} \leftarrow r_i' \vee \bigvee\{r_j | 1 \le j \le n, j \ne i\}$
5 **return** R

terminate is sufficient to verify non-termination, existing methods [10, 11, 19] typically stop once such a state is found. When returning a recurrent set, these tools often disregard its size.

However, to infer the accurate termination condition, all non-terminating states must be excluded. As presented in algorithm 1, if the recurrent set obtained is not comprehensive enough, the termination prover may fail to confirm termination, resulting in additional iterations needed to generate an adequate recurrent set. This may lead to an increase in the overall number of iterations.

To tackle this issue, we propose a recurrent set generalization scheme that considers both the number of atomic predicates and the constants in each atomic predicate. As shown in the blue comment in algorithm 1, the technique is employed after getting a recurrent set from the non-termination prover. This approach significantly reduces the number of iterations in the overall framework.

## 5.1 Framework

Recall that a recurrent set is defined as a predicate satisfying the three constraints in Definition 2.4. Given a valid recurrent set $\mathbf{R}$, our goal is to generalize $\mathbf{R}$ into a weaker recurrent set $\mathbf{R}'$. Formally, the objective is to establish that $\mathbf{R} \Rightarrow \mathbf{R}'$ holds and $\mathbf{R}'$ also satisfies all constraints in Definition 2.4, i.e. $\mathcal{RSC}(\mathbf{R}')$ is valid.

Overall, the framework of recurrent set generalization is shown in algorithm 2. We assume the input is a DNF formula $R = r_1 \vee r_2 \vee ... \vee r_n$, where each disjunct $r_i (1 \le i \le n)$ is a conjunction of several atomic predicates corresponding to an attribute set $\mathcal{A} = \{a_1, a_2, ..., a_n\}$. The recurrent set obtained from the non-termination prover is a boolean combination of atomic predicates over $\mathcal{A}$, thus can always be transformed to a DNF formula.

In algorithm 2, each disjunct $r_i (1 \le i \le n)$ is iteratively considered for predicate elimination and constant generalization (line 2-4). Predicate elimination (line 2) simplifies $r_i$ by removing redundant atomic predicates, making it simpler and potentially weaker. Next, constant generalization (line 3) refines the simplified disjunct $r_i$ by replacing each atomic predicate $a \le c$ with $a \le c'$ where $c' \ge c$, aiming to weaken the formula as much as possible or leaving it unchanged if generalization is not feasible.

After obtaining the generalized disjunct $r_i'$, $r_i$ in $\mathbf{R}$ is replaced with $r_i'$, resulting in an updated formula $\mathbf{R}$ (line 4). This iterative update across all disjuncts produces a new recurrent set $\mathbf{R}'$, which is highly likely weaker than $\mathbf{R}$ and thus contains more program states, improving the framework's efficiency in identifying the accurate termination condition.

The generalization preserves the formula's attributes, aligning with the nature of learning algorithms like the ICE framework [21]. These algorithms iteratively select optimal attributes (e.g., those with the highest information gain), making attribute selection already well-optimized. Thus, generalizing constants within atomic predicates is more relevant.

## 5.2 Generalization Techniques

Now we discuss about the predicate elimination and constant generalization techniques in detail. Note that a recurrent set $\mathbf{R}$ satisfies the constraints $\mathcal{RC}(\mathbf{R})$, $\mathcal{GC}(\mathbf{R})$ and $\mathcal{IC}(\mathbf{R})$ in Definition 2.4, where $\mathcal{GC}(\mathbf{R})$ and $\mathcal{IC}(\mathbf{R})$ are quantified formulas. Therefore, the validity of $\mathcal{GC} \wedge \mathcal{IC}$ is equivalent to the unsatisfiability of $\neg\mathcal{GC} \vee \neg\mathcal{IC}$, i.e. the following formula $\Psi(\mathbf{R})$ is unsatisfiable:

$$\exists \vec{x}_0, \vec{x}_1, \vec{x}_1'. \neg(\mathbf{R}(\vec{x}_0) \rightarrow \mathcal{G}_{loop}(\vec{x})) \vee \neg(\mathbf{R}(\vec{x}_1) \wedge \mathcal{T}_{loop}(\vec{x}_1, \vec{x}_1') \rightarrow \mathbf{R}(\vec{x}_1)) \tag{1}$$

Thus, the formula $\mathbf{R}$ can be generalized using the minimal UNSAT core, which identifies the smallest subset of constraints that preserves the unsatisfiability of a formula.

*Predicate Elimination.* Let $\mathbf{R} \equiv r_t \vee \mathbf{R}_t$ represent the initial recurrent set, where $r_t$ is the disjunct currently under consideration for generalization, and $\mathbf{R}_t$ denotes the remaining disjuncts. Suppose $r_t = p_1 \wedge \cdots \wedge p_n$ where $p_1, \ldots, p_n$ are atomic predicates. Firstly, for predicate elimination, a boolean variable $\mathcal{P}_i$ is introduced for each predicate $p_i$, indicating whether $p_i$ is retained. The predicate $r_t$ is then modified by replacing $p_i$ with $\mathcal{P}_i \rightarrow p_i$, resulting in a new form $\mathbf{R}' : \bigwedge\{\mathcal{P}_i \rightarrow p_i | 1 \le i \le n\} \vee \mathbf{R}_t$.

When $\mathcal{P}_1 \wedge ... \wedge \mathcal{P}_n$ holds, $\mathbf{R}'$ and $\mathbf{R}$ are identical. Given that $\mathbf{R}$ is a recurrent set, the following formula is unsatisfiable:

$$\Psi(\mathbf{R}') \wedge \bigwedge_{1 \le i \le n} \mathcal{P}_i \tag{2}$$

Therefore, setting $\Psi(\mathbf{R}')$ as a hard constraint and $\mathcal{P}_1, ..., \mathcal{P}_n$ as soft constraints, an UNSAT core of Equation (1) represents a minimal subset $\mathbf{U} \subseteq \{\mathcal{P}_1, ..., \mathcal{P}_n\}$ such that $\Psi(\mathbf{R}') \wedge \bigwedge_{\mathcal{P}_i \in \mathbf{U}} \mathcal{P}_i$ remains unsatisfiable. Denote $r_u = \{p_i | \mathcal{P}_i \in \mathbf{U}\}$. Predicates in $r_u$ are critical to Equation (2)'s unsatisfiability, while others can be safely eliminated. Removing these non-essential predicates yields a generalized recurrent set $\mathbf{R}_e : (\bigwedge_{p_i \in r_u} p_i) \vee \mathbf{R}_t$. It follows directly that $\mathbf{R} \rightarrow \mathbf{R}_e$ holds, as $\bigwedge_{1 \le i \le n} p_i$ implies $\bigwedge_{p_i \in r_u} p_i$.

*Constant Generalization.* Typically, an atomic predicate may be non-removable, but its range can be generalized. To this end, a constant generalization process is introduced after predicate elimination to further expand the recurrent set by systematically relaxing the constants in atomic predicates.

Each atomic predicate $a_i \le c_i (1 \le i \le n)$ in $r_t$ defines an interval $(-\infty, c_i]$. For generalization, we establish a sequence of distances $\mathcal{D} = \{0 = d_0, d_1, d_2, ..., d_m\}$ with $d_j < d_{j+1}$ for every $0 \le j < m$. These distances create boundaries $\{c_i + d_j \mid 0 \le j \le m\}$, partitioning the number line into disjoint intervals.

To find the maximum range, *digging technique* [56] is applied. It was proposed for generalizing a single counterexample, while here we extend it to generalize a recurrent set. For each atomic predicate $p_i : a_i \le c_i$, $m + 1$ intervals are introduced as follows:

$$\mathcal{I}_{i,j} := \begin{cases} a_i > c_i + d_{j-1} \wedge a_i \le c_i + d_j, & \text{if } 1 \le j \le m \\ a_i > c_i + d_{j-1}, & \text{if } j = m + 1 \end{cases}$$

Each $\mathcal{I}_{i,j}$ represents an interval separated by $c_i, c_i+d_1, ..., c_i+d_m$. The union of all intervals forms the predicate $a_i > c_i$, the negation of the original $p_i$. The digging technique attempts to remove some of the intervals while maintaining unsatisfiability of $\Psi(\mathbf{R})$, and constructs a new recurrent set from remaining intervals.

For each $\mathcal{I}_{i,j}$, a boolean variable $\mathcal{P}_{i,j}$ is introduced to represent whether $\mathcal{I}_{i,j}$ is excluded, meaning that its negation holds. By replacing each $p_i$ with $\bigwedge_{1\le j\le m+1} \mathcal{P}_{i,j} \rightarrow \neg\mathcal{I}_{i,j}$, we obtain $\mathbf{R}'$ as $\left(\bigwedge_{1\le i\le n, 1\le j\le m+1} \mathcal{P}_{i,j} \rightarrow \neg\mathcal{I}_{i,j}\right) \vee \mathbf{R}_t$. Since for each $1 \le i \le n$, $\bigwedge_{1\le j\le m+1} \neg\mathcal{I}_{i,j}$ is equivalent to the original predicate $a_i \le c_i$, the following formula is unsatisfiable:

$$\Psi(\mathbf{R}') \wedge \bigwedge_{1\le i\le n, 1\le j\le m} \mathcal{P}_{i,j} \tag{3}$$

By considering Equation (3) with hard constraints $\Psi(\mathbf{R}')$ and soft constraints $\{\mathcal{P}_{i,j} | 1 \le i \le n, 1 \le j \le m+1\}$, a minimal UNSAT core is obtained, indicating a sequence of intervals to be excluded. For an UNSAT core $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_u\}$, $r_t$ is updated to $r'_t : \bigwedge_{i=1}^{u} \neg\mathcal{I}_i$.

Similarly, we get the generalized $\mathbf{R}_g : (\bigwedge_{i=1}^{u} \neg\mathcal{I}_i) \vee \mathbf{R}_t$. Since $\bigwedge_{1\le i\le n, 1\le j\le m+1} \neg\mathcal{I}_{i,j} \Leftrightarrow r_t$ holds, the states expressed by $\mathbf{R}$ are included in thoses indicated by $\mathbf{R}_g$, thus $\mathbf{R} \Rightarrow \mathbf{R}_g$ is valid.

*Example 5.1.* Consider the program in Figure 1 again. In the first iteration, the original recurrent set $\mathbf{R}_1$ obtained is $(w \le 0 \wedge -w \le 0 \wedge t \le -2) \vee (w \le 0 \wedge -w \le 0 \wedge -2 < t \wedge 0 < t)$. The two disjuncts are considered separately. For the first disjunct, no redundant predicates are identified during predicate elimination. Using the distance set $\mathcal{D} = \{0, 1, 2, 4, 8, 10\}$, constant generalization expands the atomic predicate $t \le -2$ to $t \le -1$, yielding $w \le 0 \wedge -w \le 0 \wedge t \le -1$. For the second disjunct, predicate elimination removes the redundant predicate $-2 < t$, while constant generalization cannot expand the bounds further, yielding $w \le 0 \wedge -w \le 0 \wedge 0 < t$. Consequently, we obtain the generalized recurrent set $\mathbf{R}'_1 : (w \le 0 \wedge -w \le 0 \wedge t \le -1) \vee (w \le 0 \wedge -w \le 0 \wedge 0 < t)$, which simplifies to $w = 0 \wedge (t \le -1 \vee 0 < t)$ as discussed in Section 3. After nine iterations with our generalization technique, the accurate termination condition $\varphi^* : (w \ge 0 \wedge -w \le t \le w) \vee t = 0 \vee (w \le -2 \wedge t = 1)$ is successfully derived. Upon termination verification, we obtain $\varphi^*$ as the final result.

## 5.3 Correctness

This section establishes the correctness of algorithm 2, demonstrating that recurrent set generalization preserves non-termination as well as the key properties of algorithm 1. First, we prove that the generalized recurrent set remains a recurrent set for the loop:

Theorem 5.2. *Suppose* $\mathbf{R}$ *is a recurrent set for a loop* $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$*, then the formula* $\mathbf{R}'$ *returned by algorithm 2 is also a recurrent set for* $L$*.*

Proof. First, since $\mathbf{R}$ is a recurrent set, it follows from Definition 2.4 that there exists states $\vec{x}_0, \vec{x}'_0$ such that $\mathcal{T}_{stem}(\vec{x}_0, \vec{x}'_0) \wedge \mathbf{R}(\vec{x}'_0)$ holds. Additionally, in the process of atomic predicate elimination and constant generalization, it have been demonstrated that the generalized predicate includes all states corresponding to the origin predicate. Consequently, $\mathbf{R} \Rightarrow \mathbf{R}'$ holds, which implies that $\mathcal{T}_{stem}(\vec{x}_0, \vec{x}'_0) \wedge \mathbf{R}'(\vec{x}'_0)$ holds as well, ensuring that $\mathbf{R}'$ is reachable from the initial state $\vec{x}_0$ and $\mathcal{RC}(\mathbf{R}')$ is valid.

**Table 1: Comparative results of CondTerm and Acabar on non-terminating (NT) and terminating (T) benchmarks. The numbers in parentheses indicate cases where the accurate termination conditions were obtained.**

| Benchmarks | NT | | T | |
|---|---|---|---|---|
| Tools | CondTerm | Acabar | CondTerm | Acabar |
| #Solved. | **78(78)** | 51(41) | **136(136)** | 64(56) |
| #Both Sol. | 39(39) | | 57(49) | |
| #Unique Sol. | **39(39)** | 12(2) | **79(87)** | 7(7) |
| Avg. T. on Sol.(s) | 8.01 | **0.81** | 14.89 | **3.57** |

Second, the unsatisfiability of the formula $\Psi(\mathbf{R})$ shown in Equation (1) encodes the validity of $\mathcal{GC}(\mathbf{R})$ and $\mathcal{IC}(\mathbf{R})$ in Definition 2.4. By the properties of the UNSAT core, each step of the generalization process preserves the unsatisfiability of $\Psi$. Therefore, $\Psi(\mathbf{R}')$ remains unsatisfiable and thus $\mathcal{GC}(\mathbf{R}')$ and $\mathcal{IC}(\mathbf{R}')$ are valid.

Thus far, we have shown that $\mathcal{RC}(\mathbf{R}')$, $\mathcal{GC}(\mathbf{R}')$ and $\mathcal{IC}(\mathbf{R}')$ are valid, implying the validity of $\mathcal{RSC}(\mathbf{R}') := \mathcal{RC}(\mathbf{R}') \wedge \mathcal{GC}(\mathbf{R}') \wedge \mathcal{IC}(\mathbf{R}')$. By Definition 2.4, $\mathbf{R}'$ is a valid recurrent set for $L$. □

Next, we show that the generalization process preserves the correctness of the overall framework. In particular, with recurrent set generalization, the four theorems in Section 4.4 continue to hold.

Theorem 5.3. *With the inclusion of recurrent set generalization, algorithm 1 continues to satisfy the four theorems in Section 4.4, ensuring that the framework maintains its soundness, accurateness, termination and relative completeness.*

Proof. According to Theorem 5.2, the generalized recurrent set remains valid, thereby maintaining the soundness or relative completeness of the non-termination prover. Soundness and accurateness follow directly, since integrating recurrent set generalization with a sound non-termination prover preserves its soundness.

Termination and relative completeness are more evolved. Let $\varphi_m$ denote the desired accurate termination condition. Suppose that in the current iteration, algorithm 1 obtains a recurrent set $\mathbf{R}$ from the non-termination prover and generalizes it to $\mathbf{R}'$. Note that the generalization process may modify constants in the formula, meaning that $m(\mathbf{R}') \le m(\neg\varphi_m)$ may no longer hold. However, all states in $\mathbf{R}'$ will be excluded in the next iteration. Since $\mathbf{R} \Rightarrow \mathbf{R}'$, the states in $\mathbf{R}$ will be eliminated as well, and the non-termination prover will not infer the same recurrent set again. Therefore, as long as each recurrent set $\mathbf{R}$ obtained from the non-termination prover satisfies $m(\mathbf{R}) \le m(\neg\varphi_m)$, the search space remains finite and the algorithm is guaranteed to terminate. Consequently, the relative completeness is preserved as well. □

# 6 EVALUATION

We implement a prototype tool called CondTerm[1], which builds on the data-driven termination prover ddlTerm [55] and the data-driven non-termination prover RSLearn [26].

Following RSLearn, we employ the octagonal domain for recurrent set learning, resulting in conditions represented as boolean

[1]The tool is available at: https://doi.org/10.5281/zenodo.16891884

combination of atomic predicates in the form $\pm x \pm y \leq c$ where $x, y$ are program variables. For the digging technique in recurrent set generalization, we use distance values $\mathcal{D} = \{0, 1, 2, 4, 6, 8, 10, 20, 50\}$.

*Benchmarks.* We collect terminating benchmarks from [55] and non-terminating benchmarks from [26]. Specifically, [55] uses 170 terminating programs taken from [18] and we take all of them. In [26], there are 111 linear non-terminating programs taken from the *C-Integer Programs* category of the Termination and Complexity Competition (TermComp [24]). Programs containing multiple loops or non-deterministic loops, which are not supported by ddlTerm, are excluded, resulting in a total of 96 benchmarks. In sum, our benchmark set consists of 266 programs, including 170 terminating programs and 96 non-terminating programs.

*Environment.* All experiments are conducted on a computer with Intel (R) Core (TM) i5-12400 CPU (2.50 GHz) and 32.0 GB memory, running Ubuntu 22.04 platform. The timeout settings for ddlTerm and RSLearn follow the default configurations outlined in their respective papers and associated artifacts. Specifically, the timeout for RSLearn is set to 60 seconds, while the timeouts for the three proving strategies of ddlTerm are set to 10, 25, and 40 seconds, respectively. A result is reported as *timeout* if the tool fails to produce a result after 15 iterations.

Moreover, since termination verification can be time-consuming, two optimizations have been implemented in our tool. On one hand, a preliminary testing phase is introduced before performing termination verification. During this phase, the program is executed with randomly generated inputs. If the number of tests resulting in timeout exceeds the number of terminating tests, the tool will proceed directly to non-termination verification. Meanwhile, the traces obtained during the testing phase are incorporated into the data storage for future reuse. If no recurrent set is found, termination verification is then used as a fallback. On the other hand, recognizing that ddlTerm implements three strategies, the tool begins by attempting only the first strategy, which has a 10-second timeout. If this strategy fails, the tool proceeds directly to non-termination verification. If that also fails, the remaining two strategies are then attempted.

## 6.1 Overall Evaluation

Since termination is generally undecidable, verifying termination or non-termination is inherently challenging, let alone generating termination conditions. Despite its fundamental importance in program analysis, relatively few tools have been designed specifically for termination condition inference, far less on ensuring accuracy. Among existing tools, Acabar [20] is the most recent and most comparable one, which aims to identify terminating states as comprehensively as possible and guarantees the generation of a termination condition. Thus, we evaluate our tool CondTerm against Acabar and present the overall results in this experiment.

In Acabar, the generation of termination conditions is based on transition invariants [46]. By incrementally identifying disjunctively well-founded relations, program transitions are decomposed into those definitely terminate and those with unknown terminating behavior. A termination condition is produced when no further decomposition is possible. Since Acabar requires a specific syntax for input, we manually converted each program for the experiments. The timeout for Acabar was set to 300 seconds.

The overall results are summarized in Table 1. Out of the total 266 programs, CondTerm successfully derived accurate termination conditions for 214 programs, which is **86%** more than Acabar. Specifically, for the 96 non-terminating benchmarks, CondTerm accurately solved 78 cases, representing a **90%** increase over Acabar. Among the 170 terminating benchmarks, CondTerm obtained the accurate termination condition (i.e., *true*) for 136 programs, more than **twice** the number accurately solved by Acabar.

Notably, Acabar is primarily designed for simple single loops without stems and lacks support for non-linear assignments. Thus, there are 137 benchmarks that Acabar could not process, whereas CondTerm is able to handle and execute them without issue. Moreover, while CondTerm consistently produces accurate termination conditions, Acabar does not guarantee this level of precision.

Furthermore, we analyze the results for non-terminating and terminating benchmarks separately. For the 96 non-terminating benchmarks, CondTerm successfully solved 78 cases, obtaining the accurate termination condition for each. Acabar solved 51 cases, with 41 resulting in accurate termination conditions. Both tools achieved accurate conditions for the 39 benchmarks they solved in common. Additionally, there were 2 programs for which only Acabar produced accurate termination conditions. For these benchmarks, CondTerm have generated the same conditions, but due to limitations of the termination prover, ddlTerm was unable to verify termination, resulting in an *unknown* outcome.

Note that for non-terminating programs solved by Acabar, most results were accurate. In essence, Acabar incrementally identifies terminating states by synthesizing disjunctively well-founded candidates. For those cases, accurate termination conditions can be directly derived from these well-founded candidates, which explains why Acabar occasionally produces accurate results. However, it does not explicitly report accuracy. Furthermore, this approach would need to verify the universal non-termination of the remaining states to ensure accuracy, which is currently impractical. As a result, Acabar struggles to guarantee or report accuracy, whereas CondTerm provides a formal guarantee of precision.

Among the 170 terminating benchmarks, CondTerm solved 136 cases, all of which yielded the accurate termination condition *true*. In contrast, Acabar solved 64 cases, with 56 of these yielding the accurate condition *true*. For the 57 benchmarks solved by both tools, CondTerm produced a more accurate result in 8 cases. In these cases, Acabar returned a condition involving program variables, while CondTerm achieved the accurate condition *true*. These results highlight CondTerm's significant advantage in both the number of solved cases and the precision of termination conditions.

In terms of runtime, CondTerm is somewhat slower than Acabar, mainly due to the time-consuming (non-)termination verification required in each iteration. In fact, the runtime of CondTerm largely depends on the (non-)termination provers. Changing the provers or adjusting their timeout settings could lead to different runtimes. Meanwhile, Acabar is tailored for simpler loops and employs specialized and time-efficient algorithms for these cases.

**Table 2: Comparative results of multi-way data sharing.**

| Benchmarks | NT | | T | |
|---|---|---|---|---|
| Enable M.D.S | Y | N | Y | N |
| #Solved. | **78** | 78(0) | **136** | 134(-2) |
| T. on Sol.(s) | **625** | 673 | 2025 | **1934** |
| Avg. T. on Both Sol.(s) | **8.01** | 8.62(7.6%) | **14.03** | 14.44(2.9%) |
| Avg. Iter. on Both Sol. | **2.29** | 2.32(1.3%) | 1 | 1 |

**Table 3: Comparative results of recurrent set generalization.**

| Strategy | | With R.S.G | Without R.S.G |
|---|---|---|---|
| #Solved. | | **78** | 76(-2) |
| Avg. T.(s) | Overall | **39.11** | 54.86(40.3%) |
| | Both Solved. | **7.85** | 9.37(19.4%) |
| | Both Unknown. | **172.84** | 193.21(11.8%) |
| Avg. Iter. | Overall | **3.03** | 3.53(16.5%) |
| | Both Solved. | **2.26** | 2.45(8.4%) |
| | Both Unknown. | **4.33** | 5.20(20.1%) |

## 6.2 Evaluation of Proposed Techniques

To further validate our approach, we conduct experiments to evaluate the effectiveness of multi-way data sharing and recurrent set generalization.

*Evaluation of Multi-way Data Sharing.* To assess the impact of multi-way data sharing (M.D.S), we compare its effectiveness on non-terminating (NT) and terminating (T) benchmarks, as summarized in Table 2. The columns labeled "Y" and "N" indicate whether M.D.S is enabled or disabled. Each column reports the number of solved benchmarks, the total time spent on these benchmarks, as well as the average time and iteration counts for solved cases. For non-terminating benchmarks, disabling M.D.S results in a **7.6%** increase in total time cost and a **1.3%** increase in the number of iterations. For terminating benchmarks, CondTerm successfully solves two additional benchmarks when M.D.S is enabled, while disabling M.D.S leads to a **2.9%** increase in the average time of solved cases. These results demonstrate that multi-way data sharing effectively improves performance by reducing both time cost and iteration counts, thereby enhancing the efficiency of the overall process.

*Evaluation of Recurrent Set Generalization.* Since the recurrent set generalization is applied only after obtaining the program's recurrent set, it does not affect terminating programs. Therefore, we evaluate its effectiveness on the 96 non-terminating benchmarks. The results are presented in Table 3. The colomn "With R.S.G" and "Without R.S.G" represent the outcomes with and without recurrent set generalization, respectively. Each colomn displays the number of solved benchmarks and the average time as well as iteration counts for all benchmarks, solved benchmarks, and benchmarks reported as *unknown*.

With the inclusion of recurrent set generalization, CondTerm solved two additional benchmarks which would result in *timeout* without it. Additionally, "Without R.S.G." shows a **40.3%** increase in the average time and a **16.5%** increase in the number of iterations across all benchmarks. For benchmarks solved by both strategies, "Without R.S.G." increases average runtime by **19.4%** and iteration counts by **8.4%**. For benchmarks labeled as *unknown* by both strategies, "Without R.S.G." results in a **11.8%** increase in runtime and a **20.1%** increase in iteration counts. These results indicate that the recurrent set generalization technique significantly accelerates the iteration process and reduces the number of iterations. For solvable benchmarks, it facilitates faster convergence to the accurate termination condition, and for unsolvable benchmarks, it enables a quicker determination of the "*unknown*" result.

## 7 RELATED WORK

*Conditional termination analysis.* To date, research on generating termination conditions remains relatively limited. The earliest approach, proposed by [12], introduces potential ranking functions that are bounded but not definitely decreasing, deriving termination conditions for sequential arithmetic programs by identifying when these functions become valid. Subsequent works, such as [5] and [20], generate termination conditions by over-approximating non-terminating states, relying on algebraic analysis that restricts their applicability. Specifically, [5] focuses on difference bounds, octagonal relations and linear affine relations, while [20] is limited to linear simple loops that fit specific semantics. In contrast, our approach is independent of any specific (non-)termination provers and allows for the integration of various proving methods. By leveraging two data-driven provers, it benefits from black-box learning, allowing the learner to operate independently of the program's semantics. This flexibility enables our method to handle complex constructs such as non-linear assignments, which pose challenges for traditional algebraic methods. As a result, our approach imposes fewer structural restrictions on the programs and applies to a broader range of scenarios.

Additionally, some researchers have introduced conditional termination analysis to enhance (non-)termination analysis. The authors of [58] apply techniques from linear dynamical systems to develop a monotone conditional termination analysis. The work [4] employs a constraint-based method, while [36] relies on Hoare-style verification with second-order constraints. Other approaches include conflict-driven learning [17], policy iteration [42], and abstract interpretation [15, 51, 53, 54]. However, none of these existing methods guarantee the precision of the generated conditions. In contrast, our method explicitly ensures accuracy while maintaining soundness, termination, and relative completeness, providing a strong theoretical foundation.

*Termination analysis.* Termination analysis has been extensively studied, leading to the development of numerous tools, including Terminator and its successor T2 [7, 13, 14], Tan [32], HipTNT+ [36], Ultimate Automizer [28], and ddlTerm [55]. Termination proofs typically rely on identifying a termination argument, such as a ranking function or a loop bound. Many approaches rely on constraint solving [2, 6, 27, 34, 39, 45, 47], utilizing SMT solvers to discover such arguments. For instance, the work [39] introduces templates for ranking functions and a constraint-based synthesis of termination arguments. Some methods employ the "guess-and-check" technique, iteratively generating and verifying potential

arguments [18, 35, 52], such as DynamiTe [35], which combines static and dynamic analysis to learn ranking functions from sampled executions. Moreover, machine learning techniques have recently gained traction in termination verification [44, 55], with approaches leveraging Support Vector Machine (SVMs) [41, 57] and neural networks [1, 22, 50] to derive termination arguments. What's more, size-change termination (SCT) was introduced in [38], which is a property strictly stronger than termination [29]. SCT abstracts programs as "size-change graphs" that track monotone decreases across function calls, ensuring any infinite execution induces infinite descent in a well-founded measure. Subsequent work has extracted explicit ranking functions from SCT instances [37] and enforced termination dynamically via run-time contracts [43].

*Non-termination analysis.* Non-termination analysis was first addressed in [25], which utilizes constraint-solving techniques to identify a recurrent set. Subsequently, the authors of [10] enhance the concept and introduced the notion of closed recurrent sets. Numerous non-termination verification methods have since been developed, many based on white-box analysis. For example, the work [40] proposes the geometric non-termination argument, while [33] applies Max-SMT solving, [9] introduces program reversal to transform the problem into invariant generation, and [8] constructs a termination graph for analysis. Other approaches rely on recurrence relation solvers [19] or specialize in addressing nonlinear programs [11] and bit-vector programs [16]. Some methods adopt a black-box learning framework, such as [18], which aims to generate conjunctive recurrent sets, and [35], which utilizes dynamic invariant inference to generate candidate arguments. Recently, machine learning has been integrated into non-termination verification, as exemplified in [26], which proposes a black-box algorithm based on decision tree learning.

## 8 CONCLUSION

In this work, we propose a novel framework for inferring accurate termination conditions for programs. The framework leverages termination and non-termination provers to iteratively isolate non-terminating states and verify the remaining ones. We instantiate the framework with data-driven provers and develop a multi-way data-sharing mechanism to improve their interaction. Through formal proofs, we establish the soundness, accurateness, termination and relative completeness of our method. Additionally, we introduce generalization techniques for recurrent sets to enhance the efficiency of the iterative process. Experimental results demonstrate that our implementation significantly outperforms the current state-of-the-art tool, Acabar, by providing more accurate termination conditions across a comprehensive benchmark, highlighting its potential to advance program analysis with both theoretical guarantees and practical improvements.

## ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

## REFERENCES

[1] Yoav Alon and Cristina David. 2022. Using graph neural networks for program termination. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 910–921. https://doi.org/10.1145/3540250.3549095

[2] Amir M. Ben-Amram and Samir Genaim. 2013. On the linear ranking problem for integer linear-constraint loops. *SIGPLAN Not.* 48, 1 (Jan. 2013), 51–62. https://doi.org/10.1145/2480359.2429078

[3] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, Heidelberg, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[4] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

[5] Marius Bozga, Radu Iosif, and Filip Konečný. 2012. Deciding Conditional Termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, Cormac Flanagan and Barbara König (Eds.). Springer, Berlin, Heidelberg, 252–266. https://doi.org/10.1007/978-3-642-28756-5_18

[6] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Termination Analysis of Integer Linear Loops. In *CONCUR 2005 – Concurrency Theory*, Martín Abadi and Luca de Alfaro (Eds.). Springer, Berlin, Heidelberg, 488–502. https://doi.org/10.1007/11539452_37

[7] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better Termination Proving through Cooperation. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Berlin, Heidelberg, 413–429. https://doi.org/10.1007/978-3-642-39799-8_28

[8] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. 2012. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *Formal Verification of Object-Oriented Software*, Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov (Eds.). Springer, Berlin, Heidelberg, 123–141. https://doi.org/10.1007/978-3-642-31762-0_9

[9] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2021. Proving non-termination by program reversal. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3453483.3454093

[10] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, Berlin, Heidelberg, 156–171. https://doi.org/10.1007/978-3-642-54862-8_11

[11] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Disproving termination with overapproximation. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 67–74. https://doi.org/10.1109/FMCAD.2014.6987597

[12] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 328–340. https://doi.org/10.1007/978-3-540-70545-1_32

[13] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. *SIGPLAN Not.* 41, 6 (June 2006), 415–426. https://doi.org/10.1145/1133255.1134029

[14] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer, Berlin, Heidelberg, 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

[15] Patrick Cousot and Radhia Cousot. 2012. An Abstract Interpretation Framework for Termination. *SIGPLAN Not.* 47, 1 (Jan. 2012), 245–258. https://doi.org/10.1145/2103621.2103687

[16] Cristina David, Daniel Kroening, and Matt Lewis. 2015. Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 183–204. https://doi.org/10.1007/978-3-662-46669-8_8

[17] Vijay D'Silva and Caterina Urban. 2015. Conflict-Driven Conditional Termination. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 271–286. https://doi.org/10.1007/978-3-319-21668-3_16

[18] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 124–143. https://doi.org/10.1007/978-3-319-96145-3_7

[19] Florian Frohn and Jürgen Giesl. 2019. Proving Non-Termination via Loop Acceleration. https://doi.org/10.48550/arXiv.1905.11187 arXiv:1905.11187.

[20] Pierre Ganty and Samir Genaim. 2013. Proving Termination Starting from the End. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Berlin, Heidelberg, 397–412. https://doi.org/10.1007/978-3-642-39799-8_27

[21] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. *SIGPLAN Not.* 51, 1 (Jan. 2016), 499–512. https://doi.org/10.1145/2914770.2837664

[22] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. 2022. Neural termination analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/3540250.3549120

[23] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with AProVE. In *Automated Reasoning*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.). Springer International Publishing, Cham, 184–191. https://doi.org/10.1007/978-3-319-08587-6_13

[24] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 156–166. https://doi.org/10.1007/978-3-030-17502-3_10

[25] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/1328438.1328459

[26] Zhilei Han and Fei He. [n. d.]. Data-Driven Recurrent Set Learning For Non-Termination Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023-05). 1303–1315. https://doi.org/10.1109/ICSE48619.2023.00115

[27] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer, Berlin, Heidelberg, 304–319. https://doi.org/10.1007/978-3-642-15769-1_19

[28] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 797–813. https://doi.org/10.1007/978-3-319-08867-9_53

[29] Matthias Heizmann, Neil D. Jones, and Andreas Podelski. 2010. Size-Change Termination and Transition Invariants. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer, Berlin, Heidelberg, 22–50. https://doi.org/10.1007/978-3-642-15769-1_4

[30] Zachary Kincaid, Thomas Reps, and John Cyphert. 2021. Algebraic Program Analysis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 46–83. https://doi.org/10.1007/978-3-030-81685-8_3

[31] Daniel Kroening, Viktor Malík, Peter Schrammel, and Tomáš Vojnar. 2023. 2LS for Program Analysis. https://doi.org/10.48550/arXiv.2302.02380 arXiv:2302.02380

[32] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer, Berlin, Heidelberg, 89–103. https://doi.org/10.1007/978-3-642-14295-6_9

[33] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. [n. d.]. Proving Non-Termination Using Max-SMT. In *Computer Aided Verification* (Cham, 2014), Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, 779–796. https://doi.org/10.1007/978-3-319-08867-9_52

[34] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2013. Proving termination of imperative programs using Max-SMT. In *2013 Formal Methods in Computer-Aided Design*. 218–225. https://doi.org/10.1109/FMCAD.2013.6679413

[35] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020), 189:1–189:30. https://doi.org/10.1145/3428257

[36] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-Termination Specification Inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 489–498. https://doi.org/10.1145/2737924.2737993

[37] Chin Soon Lee. 2009. Ranking Functions for Size-Change Termination. *ACM Trans. Program. Lang. Syst.* 31, 3 (April 2009), 10:1–10:42. https://doi.org/10.1145/1498926.1498928

[38] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-Change Principle for Program Termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

[39] Jan Leike and Matthias Heizmann. 2015. Ranking Templates for Linear Loops. *Logical Methods in Computer Science* Volume 11, Issue 1 (March 2015). https://doi.org/10.2168/LMCS-11(1:16)2015 Publisher: Episciences.org.

[40] Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 266–283. https://doi.org/10.1007/978-3-319-89963-3_16

[41] Yi Li, Xuechao Sun, Yong Li, Andrea Turrini, and Lijun Zhang. 2019. Synthesizing Nested Ranking Functions for Loop Programs via SVM. In *Formal Methods and Software Engineering*, Yamine Ait-Ameur and Shengchao Qin (Eds.). Springer International Publishing, Cham, 438–454. https://doi.org/10.1007/978-3-030-32409-4_27

[42] Damien Massé. 2014. Policy Iteration-Based Conditional Termination and Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, Berlin, Heidelberg, 453–471. https://doi.org/10.1007/978-3-642-54013-4_25

[43] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 845–859. https://doi.org/10.1145/3314221.3314643

[44] Aditya V. Nori and Rahul Sharma. 2013. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/2491411.2491413

[45] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, Berlin, Heidelberg, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[46] A. Podelski and A. Rybalchenko. 2004. Transition Invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* 32–41. https://doi.org/10.1109/LICS.2004.1319598

[47] Andreas Podelski and Andrey Rybalchenko. 2011. Transition Invariants and Transition Predicate Abstraction for Program Termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, Berlin, Heidelberg, 3–10. https://doi.org/10.1007/978-3-642-19835-9_2

[48] J. R. Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (March 1986), 81–106. https://doi.org/10.1007/BF00116251

[49] J. Ross Quinlan. 2014. *C4.5: Programs for Machine Learning*. Elsevier.

[50] Wang Tan and Yi Li. 2021. Synthesis of ranking functions via DNN. *Neural Computing and Applications* 33, 16 (Aug. 2021), 9939–9959. https://doi.org/10.1007/s00521-021-05763-8

[51] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, Berlin, Heidelberg, 43–62. https://doi.org/10.1007/978-3-642-38856-9_5

[52] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, Berlin, Heidelberg, 54–70. https://doi.org/10.1007/978-3-662-49674-9_4

[53] Caterina Urban and Antoine Miné. 2014. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 412–431. https://doi.org/10.1007/978-3-642-54833-8_22

[54] Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

[55] Rongchen Xu, Jianhui Chen, and Fei He. 2022. Data-driven loop bound learning for termination analysis. In *Proceedings of the 44th International Conference on Software Engineering*. 499–510.

[56] Rongchen Xu, Fei He, and Bow-Yaw Wang. 2020. Interval counterexamples for loop invariant learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 111–122.

[57] Yue Yuan and Yi Li. 2019. Ranking Function Detection via SVM: A More General Method. *IEEE Access* 7 (2019), 9971–9979. https://doi.org/10.1109/ACCESS.2018.2890692 Conference Name: IEEE Access.

[58] Shaowei Zhu and Zachary Kincaid. 2021. Reflections on Termination of Linear Loops. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 51–74. https://doi.org/10.1007/

978-3-030-81688-9_3