Data-Driven Loop Bound Learning for Termination Analysis

Rongchen Xu[*]
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
xrc19@mails.tsinghua.edu.cn

Jianhui Chen[*]
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
chenjian16@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

Termination is a fundamental liveness property for program verification. A *loop bound* is an upper bound of the number of loop iterations for a given program. The existence of a loop bound evidences the termination of the program. This paper employs a reinforced black-box learning approach for termination proving, consisting of a loop bound *learner* and a validation *checker*. We present efficient data-driven algorithms for inferring various kinds of loop bounds, including *simple loop bounds*, *conjunctive loop bounds*, and *lexicographic loop bounds*. We also devise an efficient validation checker by integrating a *quick bound checking* algorithm and a *two-way data sharing* mechanism. We implemented a prototype tool called ddlTerm. Experiments on publicly accessible benchmarks show that ddlTerm outperforms state-of-the-art termination analysis tools by solving 13-48% more benchmarks and saving 40-77% solving time.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; • **Theory of computation → Logic and verification**.

KEYWORDS

Termination analysis, loop bound, data-driven approach

ACM Reference Format:
Rongchen Xu, Jianhui Chen, and Fei He. 2022. Data-Driven Loop Bound Learning for Termination Analysis. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510220

1 INTRODUCTION

Termination is a fundamental liveness property for program verification. It plays a central role in proving the total correctness of programs. A *loop bound* is an upper bound of the number of loop iterations for a given program. A validated loop bound thus evidences the program's termination. In this paper, we employ a black-box learning approach for termination proving, which consists of two components: (a) a *learner* that generates a loop bound candidate; and (b) a *checker* that either confirms the correctness of the candidate or produces a counterexample to refute the candidate.

There exist some prior works on data-driven termination proving [14, 31, 41]. The work [31] employs testing to generate data examples and then applies quadratic programming to learn the loop bound candidates. Unfortunately, this approach can only infer loop bounds in the form of simple affine functions (called *simple loop bound* in this paper). As a result, it is inapplicable to many realistic programs. The work [41] enriches the forms of the loop bounds. It learns a set of affine expressions by linear interpolation and then arranges them into piecewise, lexicographic, or multiphase forms. However, affine expressions obtained by simple combinations of data examples are imprecise in complex situations. The work [14] employs syntax-guided synthesis (SyGuS) [1] to generate a mass of expressions and then combines these expressions into the loop bound candidate. However, this approach takes into consideration only syntactical information of programs. Due to the lack of program semantics knowledge, their synthesizer can hardly infer proper loop bounds for complicated programs.

In this paper, we propose techniques to enhance both *learner* and *checker* of the data-driven loop bound learning approach. Firstly, we reinforce the learner by proposing a series of data-driven algorithms to learn various loop bounds, i.e., *simple loop bounds*, *conjunctive loop bounds*, and *lexicographic loop bounds*. Our data-driven algorithms consider program semantics. More specifically, the program states, together with their transition relations, are recorded and utilized during the algorithms. The expressibility and applicability of the loop bound approach are thus significantly enhanced. With a combination of these learning algorithms, our approach is able to prove the termination of complicated programs with non-linear loop bounds.

We propose a *quick bound checking* technique to enhance the *checker*. We observe that only in the last round does the checker validate the loop bound candidate, and in all other rounds, the checker is only responsible for providing counterexamples to refute the candidates. Given that falsification is always cheaper than verification, it is thus worthwhile to apply a quick falsification check before the complete validation check. To this end, we propose a bounded

[*]Both authors contributed equally to this research.
[†]Fei He is the corresponding author.

```
j = 0;
while(x != 0) {
  print(x, j);
  j = j + 1;
  if(x < 10)
    x = x + 1;
  else x = 0; }
```

Figure 1: An example

```
assume(i >= m(x));
while(x != 0) {
  assert(i > 0);
  i = i - 1;
  if(x < 10)
    x = x + 1;
  else x = 0; }
```

Figure 2: Validation task

model checking-based technique for quickly refuting the incorrect bound candidates.

The complete validation of a loop bound candidate can be reduced to a safety verification problem, which usually involves another learning process – the loop invariant learning. Obviously, loop bound learning and loop invariant learning are strongly related and should not be regarded as two independent processes. To this end, we propose a *two-way data sharing* mechanism between these two processes: on the one hand, the data examples in bound learning are reused in invariant learning; on the other hand, in case of safety verification failures, the generated counterexamples are reused by the bound learner to refine its bound candidates.

We implement a prototype tool called ddlTerm. We take the benchmarks from [14] to evaluate the efficiency of our approach. Compared with the state-of-the-art termination analysis tools, including AProVE [17, 18, 40], UAutomizer [9, 20], FreqTerm [14], and MuVal [24], ddlTerm solves 13-48% more benchmarks and reduces 40-77% analysis time.

To summarize, this paper makes the following contributions:

- We present a series of data-driven algorithms for inferring various loop bounds, including simple loop bounds, conjunctive loop bounds, and lexicographic loop bounds.
- We propose a quick bound checking algorithm for efficiently refuting incorrect loop bound candidates.
- We propose an efficient data sharing mechanism between bound and invariant learning.
- We implement a prototype tool and conduct experiments on publicly accessible benchmarks. Results show the outstanding performance of our approach over state-of-the-art tools.

The rest of the paper is organized as follows: Section 2 motivates our approach using a simple example. Section 3 introduces some background knowledge. Section 4 presents our data-driven loop bound learning algorithms. Section 5 employs the loop bound in termination proving. Section 6 reports evaluation results, Section 7 discusses related work and Section 8 concludes the paper.

2 OVERVIEW

We employ a simple program (in Figure 1) to show the basic idea of our approach. The original version of this program (i.e., the black codes) iteratively increases x until it equals 10 and then resets x to 0. The green codes are used for data generation, where the variable j records the current iteration number, and the `print(...)` method outputs the current values of x and j at each iteration.

Table 1: Analysis procedure of example

#	Loop-Bound	Checking	Dataset
1	$m_1(x) = 0$	cex1: $x_0 = -2$	$\mathcal{H}_a : \{\triangle\}$
2	$m_2(x) = -x$	cex2: $x_0 = 1$	$\mathcal{H}_b : \{\triangle, \diamond\}$
3	$m_3(x) = \max(-x, -x + 11)$	cex3: $x_0 = 11$	$\mathcal{H}_c : \{\triangle, \diamond, ⬠\}$
4	$m_4(x) = \max(-x + 11, 1)$	pass	-

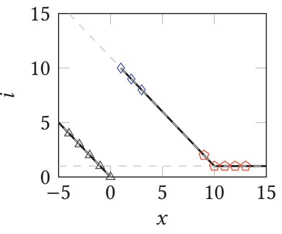

Figure 3: Loop bound and dataset

Along with a terminating execution of this program, its output is a sequence of value pairs

$$\langle x_0, 0\rangle \to \langle x_1, 1\rangle \to \cdots \to \langle x_k, k\rangle,$$

where x_i is the value of x at the i-th iteration, and k is the iteration at which the loop terminates. For each output value pair $\langle x_i, i\rangle$, let $idc_i \triangleq k - i + 1$ be the number of remaining iterations until the execution terminates, called *iteration down counter*. Each pair of x_i and idc_i is called a *data example*. We collect all data examples along this execution, and add them to a *dataset* $\mathcal{H}$, i.e.,

$$\mathcal{H} = \{\langle x_i, idc_i\rangle \mid 0 \le i \le k\}$$

We attempt to infer from $\mathcal{H}$ a symbolic *loop-bound expression* $m(x)$ that represents an upper bound candidate on the number of loop iterations (Section 4). Validation of the inferred loop-bound expression is reduced to the safety checking of the instrumented program in Figure 2 (Section 5). If the checking succeeds, the inferred loop bound is correct, and we prove the termination of the program. Otherwise, the safety checker returns a counterexample, with which we enlarge the dataset $\mathcal{H}$ and refine the inferred loop bound expression.

The analysis procedure for our motivating example is shown in Table 1. Let us start with a trivial loop bound $m_1(x) = 0$. The validation task for m_1 is to verify the program in Figure 2 with the expression `m(x)` at line 1 being replaced by `0`. This validation obviously fails, and a counterexample `cex1` with $x_0 = -2$ is returned, where x_0 is the initial value of x. We now have the information that a trace with $x_0 = -2$ can refute the loop bound m_1. We thus use the same initial value and other nearby values (by mutation test) to run the program in Figure 1 and collect the following data examples (marked as $\triangle$ in Figure 3) from its outputs:

$$\mathcal{H}_a = \{\langle -4, 4\rangle, \langle -3, 3\rangle, \langle -2, 2\rangle, \langle -1, 1\rangle, \langle 0, 0\rangle\}$$

In the remainder of this paper, we simply say that these data examples are obtained from the counterexample `cex1`.

We learn from $\mathcal{H}_a$ (by the approach in Section 4.1) a so-called *simple loop bound* $m_2(x) = -x$. Validation checking of m_2 gives a new counterexample trace cex2 with $x_0 = 1$. Similar to the first round, more data examples (denoted as $\diamond$ in Figure 3) are obtained from cex2. The dataset becomes $\mathcal{H}_b : \{\triangle, \diamond\}$. Apparently, it is impossible to find a linear expression to cover all data examples tightly in $\mathcal{H}_b$. However, we could use clustering to group $\mathcal{H}_b$ into two subsets, e.g., $\mathcal{H}_{b1} : \{\triangle\}$ and $\mathcal{H}_{b2} : \{\diamond\}$, and learn two simple loop bounds, e.g., $m_{b1} = -x$ and $m_{b2} = -x + 11$, from them respectively. The maximal operation applied to these two expressions (consider no simplification), i.e., $m_3(x) = \max(-x, -x + 11)$, gives a new loop bound, called a *conjunctive loop bound* (see Section 4.2).

The loop bound $m_3(x)$ is still not valid. Its validation checking returns a counterexample trace cex3 with $x_0 = 11$, from which we obtain more data examples (denoted as $\pentagon$ in Figure 3). The dataset is now $\mathcal{H}_c : \{\triangle, \diamond, \pentagon\}$. Let $\mathcal{H}_{c1}, \mathcal{H}_{c2}, \mathcal{H}_{c3}$ be the set of $\triangle$, $\diamond$, $\pentagon$ in $\mathcal{H}_c$, respectively. Note that $\mathcal{H}_{c3}$ contains an exceptional case (i.e., its leftmost example), which may affect the precision of the learning result. To address this problem, we propose a greed tactic to learn a set of so-called *close loop bound*s, which characterize *local features* of the subsets. For example, the close loop bounds learned from $\mathcal{H}_{c3}$ could be $m(x) = 1$ and $m(x) = 2$. By combining all learned loop bounds together, we get $m_4(x) = \max(-x, -x + 11, 1, 2)$. Finally, we use a set covering-based technique to sift redundant loop bounds and get $m_4(x) = \max(-x + 11, 1)$. The expression $m_4(x)$ passes the validation checking and is a correct loop bound. The termination of the program is thus proved.

3 PRELIMINARIES

3.1 Notations

We use $\mathcal{A}, \mathcal{B}, \cdots$ to denote the general sets, and $\mathbf{x}, \mathbf{y}, \cdots$ to denote vectors. We denote $\mathbf{x}[i]$ the i-th component of $\mathbf{x}$, and $\mathbf{x} \cdot \mathbf{y}$ the scalar product of $\mathbf{x}$ and $\mathbf{y}$. Denote $\mathbb{N}$ the set of natural numbers.

Given a binary relation $\succ$ on a set $\mathcal{W}$, we say $a \in \mathcal{W}$ is a *least element* w.r.t. $\succ$, if $a \not\succ b$ for any $b \in \mathcal{W}$. We say $\succ$ is a *well-founded relation* on $\mathcal{W}$, if every non-empty subset of $\mathcal{W}$ has a least element w.r.t. $\succ$. The set $\mathcal{W}$, together with the well-founded relation $\succ$, is then called a *well-ordered* set. A *chain* is a sequence of elements $e_1, e_2, \cdots, e_n \in \mathcal{W}$ such that $e_i \succ e_{i+1}$ for $i = 1, 2, \cdots, n-1$. Given an element $e \in \mathcal{W}$, we denote $|e|$ the *length* of the longest chain in $\mathcal{W}$ starting at e. A *well-ordered* set has no infinite chain. Let $\succcurlyeq$ be the reflexive closure of $\succ$, which is not well-founded. For example, $\mathbb{N}$ is a well-ordered set w.r.t the greater-than relation $>$, its least element is 0, and there is no infinite chain from any positive number in $\mathbb{N}$. The reflexive closure of $>$ is $\geq$.

An *affine function* is composed of a linear function and a constant, e.g., $f(x, y) = ax + by + c$. A function $f(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$ is said *convex* if $f(t \cdot \mathbf{x}_1 + (1-t) \cdot \mathbf{x}_2) \leq t \cdot f(\mathbf{x}_1) + (1-t) \cdot f(\mathbf{x}_2)$ for any $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}$ and $t \in [0, 1]$.

3.2 Convex Optimization and Set Covering Problem

Given an *objective function* $f(\mathbf{x})$, the *convex optimization problem* [5] is to find an optimum $\mathbf{x}^*$ that minimizes $f(\mathbf{x})$ and satisfies all constraints, i.e.,

$$\arg\min_{\mathbf{x}} f(\mathbf{x}) \quad s.t. \quad \bigwedge_{i=1}^{m} g_i(\mathbf{x}) \geq 0$$

where f and $g_i (1 \leq i \leq m)$ are all convex functions.

Let $\mathcal{U}$ be a universe, and $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be the set of its subsets. Define a *selector variable* X_i (either 1 or 0) for each S_i in $\mathcal{S}$, representing if this subset is selected or not. Assume each S_i is associated with a *cost* C_i, the *set covering problem* [8] is to

$$\arg\min_{\mathbf{X}} \sum_{i=1}^{k} C_i \cdot X_i \quad s.t. \quad \mathcal{U} = \bigcup_{X_i = 1} S_i$$

3.3 Clustering

The clustering problem is to divide a set of objects into several subsets, called *cluster*s, such that similar objects are more likely to be assigned to the same cluster. There are many different clustering models, e.g., centroid-based [30, 39], density-based [2, 13], etc. This paper is mainly focused on the *centroid-based clustering*, where each cluster is represented as a central vector. The most widely-used centroid-based clustering algorithm is *k-means* [30]. Given a set of observations $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$, where each observation is a vector, k-means clustering aims to partition the n observations into k $(\leq n)$ subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares

$$\arg\min_{\mathcal{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mu_i\|^2$$

where μ_i is the mean of points in S_i.

4 DATA-DRIVEN LOOP BOUND LEARNING

This section discusses the learning of *loop bounds* from a given *dataset*. To simplify the discussions, we assume the input program contains a single loop [1].

A *program state* is a valuation of the program variables, usually represented as a vector of values. Let $\mathcal{X}$ be the set of reachable program states at the loop header, and $\mathcal{W}$ a well-ordered set w.r.t. a well-founded relation $\succ$.

Definition 4.1. A *loop bound* is a function $\mathbf{m}$ that maps $\mathcal{X}$ into $\mathcal{W}$ such that from any state $\mathbf{x} \in \mathcal{X}$, the number of remaining iterations until termination is no more than $\mathbf{m}(\mathbf{x})$.

The loop bound is originally unknown. However, we can extract some data examples from the program in the form of $\langle \mathbf{x}, \mathbf{idc} \rangle$, where $\mathbf{x}$ is a program state in $\mathcal{X}$, and $\mathbf{idc} \in \mathcal{W}$ represents the iteration down counter at that state. After we collect a sufficient number of data examples, we are capable of inferring the loop bound function from these examples.

We use BoundLearn to represent any procedure of *loop bound learning*. It should satisfy the following definition:

Definition 4.2. Given a dataset $\mathcal{H}$, the BoundLearn procedure outputs a *loop bound candidate* $\mathbf{m}(\mathbf{x})$ such that

$$\forall \langle \mathbf{x}, \mathbf{idc} \rangle \in \mathcal{H}.\ \mathbf{m}(\mathbf{x}) \succcurlyeq \mathbf{idc}$$

[1]Programs containing multiple or nested loops can be handled by the techniques introduced in [16].

4.1 Simple Loop Bound Learning

Recall that a loop bound $\mathbf{m}(\mathbf{x})$ is a function over an n-dimensional vector $\mathbf{x}$. A *simple loop bound* uses $\mathbb{N}$ as its well-ordered set and can be expressed in an *affine function*. In this case, the iteration down counter **idc** also belongs to $\mathbb{N}$, denoted as *idc*. We employ *affine templates*, e.g., $m(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} + b$, to instantiate the simple loop bound candidates. More specifically, an affine function $m(\mathbf{x})$ with undetermined coefficients can be expressed as a *meta-function*

$$\bar{m}(\hat{\mathbf{a}}, \mathbf{x}) = \sum_{i=1}^{n} \hat{\mathbf{a}}[i] \cdot \mathbf{x}[i] + \hat{\mathbf{a}}[n+1],$$

where $\hat{\mathbf{a}}$ is an n+1-dimensional vector representing the undetermined coefficients. Once $\hat{\mathbf{a}}$ is determined, the meta-function $\bar{m}(\hat{\mathbf{a}}, \mathbf{x})$ becomes an affine function $m(\mathbf{x})$.

Given a dataset $\mathcal{H}$, the simple loop bound learning task can be formalized as the following *convex optimization problem* over the vector space of the coefficients $\hat{\mathbf{a}}$:

$$\min \ vCost(\mathcal{H}, \hat{\mathbf{a}}) + mCost(\hat{\mathbf{a}}) \tag{1}$$
$$s.t. \bigwedge_{\langle \mathbf{x}, idc\rangle \in \mathcal{H}} \bar{m}(\hat{\mathbf{a}}, \mathbf{x}) \geq idc$$

Many existing techniques for the convex optimization with constraints (e.g., COBYLA method [36] and SLSQP method [21]) can be used as `SimpleBoundLearn`.

Recall that convex optimization requires its *objective function* to be a convex function. The cost functions $vCost(\mathcal{H}, \hat{\mathbf{a}})$ [31] and $mCost(\hat{\mathbf{a}})$ should be convex. We design these two cost functions based on the following considerations. First, a good loop bound candidate should be close to the data examples. So, we define

$$vCost(\mathcal{H}, \hat{\mathbf{a}}) \triangleq \alpha_v \cdot \sum_{\langle \mathbf{x}, idc\rangle \in \mathcal{H}} \left(\bar{m}(\hat{\mathbf{a}}, \mathbf{x}) - idc\right)^2,$$

where α_v is a hyperparameter. This *vCost* definition tends to make the loop bound as close as possible to the data examples. Differing from [31], we think that a good loop bound candidate should also have a natural form. For example, it is unwise to use a big constant $\hat{\mathbf{a}}[n+1]$ to cover all data examples. To this end, we define

$$mCost(\hat{\mathbf{a}}) \triangleq \alpha_m \cdot \sum_{i=1}^{n+1} \hat{\mathbf{a}}[i]^2,$$

where α_m is also a hyperparameter. This *mCost* definition tends to make the coefficients as small as possible. A solution of the above optimization problem assigns values to the coefficients $\hat{\mathbf{a}}$ and thus produces a simple loop bound candidate $m(\mathbf{x})$.

4.2 Conjunctive Loop Bound Learning

There are programs (e.g., the motivating example in Section 2) whose loop bound cannot be caught by a simple loop bound expression. For these programs, we cannot apply `SimpleBoundLearn` on the whole dataset. Instead, the so-called *conjunctive loop bound* should be applied, which can be viewed as a *piecewise affine function* over the program variables. Algorithm 1 depicts our algorithm for learning *conjunctive loop bounds*.

Dataset Clustering. A natural idea for learning a conjunctive loop bound is to divide the dataset $\mathcal{H}$ into several subsets and learn a simple loop bound from each of these subsets. Then a combination of these simple loop bounds forms a conjunctive loop bound. For example, the dataset $\mathcal{H}_b$ in our motivating example can be partitioned into two subsets, i.e., $\mathcal{H}_{b1} : \{\triangle\}$ and $\mathcal{H}_{b2} : \{\diamond\}$ (see Figure 3). We obtain two simple loop bound candidates, i.e., $m_1 = -x$, $m_2 = -x + 11$, by applying `SimpleBoundLearn` on these two subsets, respectively. A conjunctive loop bound candidate is $m = \max(m_1, m_2)$, which is a valid bound on the domain of $x \in (-\infty, 10]$.

As presented on lines 2 to 3 in Algorithm 1, we first estimate a maximal number k of the clusters and then employ a centroid-based clustering (k-means) to partition the dataset $\mathcal{H}$ into no more than k subsets. After clustering, we are able to learn *local features* from the subsets, which can hardly be learned from the whole dataset.

Algorithm 1: `ConjunctiveBoundLearn`($\mathcal{H}$)

```
input  : A data set $\mathcal{H}$
output : A conjunctive loop bound candidate $m$
1  $\mathcal{M} \leftarrow \emptyset$
2  $k \leftarrow$ GetMaxClusterNumber($\mathcal{H}$)
3  $\mathcal{H}_1, \cdots, \mathcal{H}_k \leftarrow$ Clustering($\mathcal{H}, k$)
4  for $i \leftarrow 1$ to $k$ do
5      $m_i^s \leftarrow$ SimpleBoundLearn($\mathcal{H}_i$)
6      $m_i^{c_1} \cdots m_i^{c_n} \leftarrow$ CloseBoundLearn($\mathcal{H}_i$)
7      $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_i^s, m_i^{c_1}, \cdots, m_i^{c_n}\}$
8  $\mathcal{M}_c \leftarrow$ BoundCombine($\mathcal{M}, \mathcal{H}$)
9  return $\max(\mathcal{M}_c)$
10
11 function CloseBoundLearn($\mathcal{H}$)
12     if $\mathcal{H} \neq \emptyset$ then
13         $m \leftarrow$ SoftConvexOptimize ($\mathcal{H}$)
14         $C \leftarrow$ Covered($m, \mathcal{H}$)
15         $\mathcal{S} \leftarrow \mathcal{H} - C$
16         return $\{m\} \cup$ CloseBoundLearn($\mathcal{S}$)
17     return $\emptyset$
18
19 function BoundCombine($\mathcal{M}, \mathcal{H}$)
20     foreach $m \in \mathcal{M}$ do
21         $C_m \leftarrow$ GetCost($\mathcal{M}, \mathcal{H}$)
22         foreach $v \in \mathcal{H}$ do
23             $a_{m,v} \leftarrow$ GetCoverage($m, p$)
24     $\mathcal{M}_c \leftarrow$ SetCoveringProblem($\{C_m\}, \{a_{m,v}\}$)
25     return $\mathcal{M}_c$
```

Close Loop Bound. Although the centroid-based clustering helps to learn local features, there are still some exceptional cases. For example, the data examples around the discontinuities of $m(\mathbf{x})$, e.g., $x = 0$ and $x = 10$ in Figure 3, might be grouped into the same subset. Moreover, the counterexamples returned by the checker are often near the discontinuities. In our motivation example, the counterexamples in the second and the third rounds are very close to the discontinuities ($x = 0$ and $x = 10$). As a result, data examples obtained from these counterexamples are very likely to get clustered together. If $m(\mathbf{x})$ is continuous on the discontinuities, e.g., $x = 10$ in Figure 3, we can hardly learn a suitable loop bound candidate from these subsets. For example, in the third round of the motivation example, if we directly apply `SimpleBoundLearn` to the cluster $\mathcal{H}_{c3} : \{⬠\}$, a loop bound candidate $m_3 = -x/4 + 17/4$ could be learned. This loop bound is obviously inaccurate and can be refuted by any input $x \geq 17$.

We propose *close loop bounds* to address the above problem. Basically, a close loop bound does not require covering all data examples in $\mathcal{H}$. We introduce a *slack set* $\mathcal{S} \subseteq \mathcal{H}$ and allow the data examples in $\mathcal{S}$ to exceed the close loop bound.

A natural idea for learning a close loop bound is to drop the constraints relevant to $\mathcal{S}$ from the convex optimization equation (1). However, it is very difficult to precisely determine $\mathcal{S}$. Alternatively, we define a *slack variable* $\delta_\mathbf{x}$ for each data example $\langle \mathbf{x}, idc\rangle \in \mathcal{S}$, representing the distance of this example exceeding the loop bound candidate, i.e. $\delta_\mathbf{x} = idc - m(\mathbf{x})$. Apparently, the slack variables' values should be greater than or equal to 0. Then the close loop bound learning problem can be formalized as the following convex optimization with soft constraints (called `SoftConvexOptimize`):

$$\min\ vCost(\mathcal{H} - \mathcal{S}, \hat{\mathbf{a}}) + mCost(\hat{\mathbf{a}}) + sCost(\mathcal{S}) \tag{2}$$

$$s.t. \bigwedge_{\langle \mathbf{x}, idc\rangle \in \mathcal{H}-\mathcal{S}} \bar{m}(\hat{\mathbf{a}}, \mathbf{x}) - idc \geq 0 \quad \wedge$$

$$\bigwedge_{\langle \mathbf{x}, idc\rangle \in \mathcal{S}} \bar{m}(\hat{\mathbf{a}}, \mathbf{x}) + \delta_\mathbf{x} - idc \geq 0 \quad \wedge \bigwedge_{\langle \mathbf{x}, idc\rangle \in \mathcal{S}} \delta_\mathbf{x} \geq 0$$

Usually, we do not want too many data examples to exceed the loop bound. We thus devise the cost function

$$sCost(\mathcal{S}) = \alpha_s \cdot \sum_{\langle \mathbf{x}, idc\rangle \in \mathcal{S}} \sqrt{\delta_\mathbf{x}}\ ,$$

where α_s is a hyperparameter. This cost function encodes our preference over fewer data examples that significantly exceed the loop bound, rather than a large number of data examples that slightly exceed the loop bound.

For each subset $\mathcal{H}_i$, we learn a simple loop bound (at line 5) and a set of close loop bounds (at line 6). All of the learned simple and close loop bounds are added to $\mathcal{M}$. The `CloseBoundLearn` procedure in Algorithm 1 depicts our learning algorithm for close loop bounds. It takes a dataset $\mathcal{H}$ as the input and recursively learns close loop bound over the uncovered part until $\mathcal{H}$ is fully covered. For example, if we call `CloseBoundLearn` on $\mathcal{H}_{c3}$, we first get $m_{c3}^{(1)} = 1$, which covers the right-most four data examples in $\mathcal{H}_{c3}$. Then, the algorithm is recursively called on the uncovered data examples. A new loop bound candidate $m_{c3}^{(2)} = 2$ is produced, which covers the remaining data example in $\mathcal{H}$. It is clear that $m_{c3}^{(1)}$ characterizes the local features of the dataset $\mathcal{H}_{c3}$.

Loop Bounds Combination. After clustering and close loop bound learning, we obtain a set $\mathcal{M}$ of loop bounds. Some of the learned loop bounds may be redundant and can be safely removed. We propose an approach for generating a concise combination of learned loop bounds.

```
while(x >= 0 && y > 0)
{
  print(x, y);
  if(*) {
    println("B1");
    y = y - 1;
  } else {
    println("B2");
    y = * ;
    x = x - 1;
  }
}
```

Figure 4: Lexico. example

```
assume(i1 >= M1(x,y));
assume(i2 >= M2(x,y));
while(x >= 0 && y > 0) {
  assert(i2 > 0);
  if(i1 > 0) {
    i1 = i1 - 1; }
  else { i2 = i2 - 1;
    i1 = *;
    assume(i1 >= M1(x,y)); }
  if(*) { y = y - 1; }
  else { y = *; x = x - 1; }
}
```

Figure 5: Lexico. validation task

For each loop bound candidate $m \in \mathcal{M}$, we introduce a variable C_m to represent its cost and a Boolean variable $X_m \in \{0, 1\}$ to indicate if m is kept. The kept loop bounds should cover all data examples. We define an indicator variable $a_{v,m}$ for each data example $v \in \mathcal{H}$. The variable $a_{v,m} = 1$ means that the data example $v : \langle \mathbf{x}, idc\rangle$ is covered by m, i.e., $m(\mathbf{x}) \geq idc$. The loop bounds combining task can then be formalized as the following *set covering problem* with minimum cost:

$$\min \sum_{m \in \mathcal{M}} C_m \cdot X_m \tag{3}$$

$$s.t. \bigwedge_{v \in \mathcal{H}} \left(\sum_{m \in \mathcal{M}} a_{v,m} \cdot X_m \geq 1 \right) \wedge \bigwedge_{m \in \mathcal{M}} X_m \in \{0, 1\}$$

Note that both $a_{v,m}$ and C_m can be calculated beforehand. After solving this optimization problem, we get a set of selected loop bound candidates $\mathcal{M}_c = \{m \mid m \in \mathcal{M} \wedge X_m = 1\}$.

The `BoundCombine` procedure is presented in lines 19 to 25 of Algorithm 1. We first calculate the costs and the coverage indicators and then solve the corresponding set covering problem to get the reduced loop bound set $\mathcal{M}_c$. In lines 8 to 9, we call this procedure and return the combination of the pruned bound candidates. Let us continue the verification of our motivation example on the third round, the set of loop bound candidates learned from $\mathcal{H}_c$ is $\mathcal{M} = \{-x, -x + 11, 1, 2\}$. After `BoundCombine`, a more concise set $\mathcal{M}_c = \{-x + 11, 1\}$ is produced. Combining these candidates together, we get $m = \max(-x + 11, 1)$, which is a correct loop bound, being able to prove the termination of the example program.

4.3 Lexicographic Loop Bound Learning

The conjunctive loop bound is still powerless in handling programs with rather complicated control-flows. This section proposes a new method to learn *lexicographic loop bound* (*LexLB*), composed of simple and conjunctive loop bounds but with more powerful expressibility. In fact, the simple and conjunctive loop bounds can be regarded as special cases ($n = 1$) of *n-dimensional lexicographic loop bound* (*n-LexLB*).

Lexicographic Loop Bound. We employ a new example program (in Figure 4) to illustrate the lexicographic loop bound learning. The symbol `*` in the program denotes a non-deterministic value (`bool` or `int`). Note that the branches of the `if-then-else` statement may interleave arbitrarily across iterations, the maximal number of iterations of this loop can reach $O(x_0 \cdot y_{max})$, where x_0 is the initial value of x and y_{max} is the maximum value that y may attain during the whole program execution. From this perspective, the loop bound of this program should be expressed in a non-linear expression. However, as we commonly know that learning a non-linear loop bound requires *general non-linear optimization*, and validating a non-linear loop bound requires *non-linear constraint solving*. In general, optimizing and validating non-linear loop bounds are both undecidable.

Instead, we adopt the *lexicographic loop bound*, which is an n-dimensional function over the program variables, i.e.,

$$\mathbf{m}(\mathbf{x}) \triangleq \langle m_1(\mathbf{x}), \cdots, m_n(\mathbf{x})\rangle.$$

For example, $\mathbf{m}(x, y) = \langle y, x\rangle$ is a 2-*LexLB* of the program in Figure 4. The lexicographic loop bound function maps the program states into a well-ordered set formed by an n-dimensional vector space under a *lexicographic order*.

To learn a lexicographic loop bound, we need to address the following two problems: (1) how to decide the dimensionality of the lexicographic function? (2) how to decide the lexicographic order over these dimensions?

A lexicographic loop bound should cover all paths within the loop body. Let k be the number of paths in the loop body, and n be the assumed dimensionality of the lexicographic function. We consider all solutions of binding the k paths to the n dimensions, where each dimension is bound to at least one path. Each solution gives a lexicographic order. A solution is said *feasible* if from which we are able to learn a valid loop bound. If there are multiple feasible solutions, we evaluate their learned bounds in each dimension and choose the best one. The dimensionality of the lexicographic function is initialized to 2 and then progressively increased until either we find a feasible solution or a predefined dimensionality bound k_0 ($k_0 \le k$) is reached.

Extract Data Examples. We employ the program in Figure 4 to illustrate the extraction of data examples. The method can be naturally extended to general cases.

Considering the instrumented codes (the green codes) in Figure 4, we use B1 and B2 to label the branch executed in each loop iteration. The program's output is a sequence of tuples; each tuple corresponds to a loop iteration and is in the form of $\langle x, y, lb\rangle$, where x, y are variables' values, and lb is the label of the executed branch. For example, given an input of $x = 2$ and $y = 1$, a (possible) output of this program is:

$$\rho : \langle 2, 1, \mathtt{B2}\rangle \rightarrow \langle 1, 1, \mathtt{B1}\rangle \rightarrow \langle 1, 0, \mathtt{B2}\rangle \rightarrow \langle 0, 2, \mathtt{B1}\rangle \rightarrow \langle 0, 1, \mathtt{B1}\rangle$$

Denote $\mathcal{H}_\rho$ the set of all outputted traces of this program.

The loop body of this program contains two paths (i.e., the two branches). Suppose we want to learn a 2-dimensional *LexLB*, e.g., $\mathbf{m}(x, y) = \langle m_1, m_2\rangle$. We first introduce a 2-dimensional iteration down counter $\mathbf{idc} = \langle idc_1, idc_2\rangle$. Without loss of generality, suppose branch B1 is bound to idc_1 and branch B2 is bound to idc_2. For each trace in $\mathcal{H}_\rho$, we calculate the values of $\mathbf{idc}$ backward along that trace. Let us consider the trace ρ for example, the value of $\langle idc_1, idc_2\rangle$ is $\langle 1, 0\rangle$ at its last tuple $\langle 0, 1, \mathtt{B1}\rangle$ (the branch B1 is passed). We lexicographically increase $\langle idc_1, idc_2\rangle$ backward along ρ: if the passed tuple contains the B1 label, we increase idc_1 by 1; otherwise, if it contains the B2 label, we increase idc_2 by 1 and reset

Algorithm 2: LexicoBoundLearn($\mathcal{H}, n$)

```
input  : A dataset $\mathcal{H}$, a number of dimension $n$
output : An $n$-dimensional lexicographic loop bound
         candidate $\mathbf{m}$
1 $\mathcal{H}_1, \cdots, \mathcal{H}_n \leftarrow \emptyset, \cdots, \emptyset$
2 foreach $\langle \mathbf{x}, \mathbf{idc}\rangle \in \mathcal{H}$ do
3     foreach $idc_i \in \mathbf{idc}$ do
4         $\mathcal{H}_i = \mathcal{H}_i \cup \{\langle \mathbf{x}, idc_i\rangle\}$
5 for $i = 1$ to $n$ do
6     $m_i \leftarrow$ Simple/ConjunctiveBoundLearn($\mathcal{H}_i$)
7 $\mathbf{m} \leftarrow \langle m_1, \cdots, m_n\rangle$
8 return $\mathbf{m}$
```

idc_1 to 0. Finally, we get a sequence σ of 2-dimensional iteration down counters:

$$\sigma : \langle 0, 2\rangle \xleftarrow{\mathtt{B2}} \langle 1, 1\rangle \xleftarrow{\mathtt{B1}} \langle 0, 1\rangle \xleftarrow{\mathtt{B2}} \langle 2, 0\rangle \xleftarrow{\mathtt{B1}} \langle 1, 0\rangle$$

Each counter of σ corresponds to a tuple of ρ. Pairing the program state in each tuple of ρ and the corresponding iteration down counter of σ gives a set of data examples:

$$\mathcal{H}_{\langle\langle x,y\rangle, \langle idc_1, idc_2\rangle\rangle} = \left\{\begin{matrix}\langle\langle 2,1\rangle, \langle 0,2\rangle\rangle, \langle\langle 1,1\rangle,\langle 1,1\rangle\rangle, \langle\langle 1,0\rangle, \langle 0,1\rangle\rangle \\ \langle\langle 0,2\rangle, \langle 2,0\rangle\rangle,\langle\langle 0,1\rangle, \langle 1,0\rangle\rangle\end{matrix}\right\}$$

Learn Lexicographic Loop Bound. Algorithm 2 presents our learning algorithm for n-dimensional lexicographic loop bounds. For each data example $\langle \mathbf{x}, \mathbf{idc}\rangle$ in $\mathcal{H}$, we combine its program state $\mathbf{x}$ with each dimension idc_i of $\mathbf{idc}$. The result forms a set of data examples $\langle \mathbf{x}, idc_i\rangle$ with 1-dimensional counters. We add the data example $\langle \mathbf{x}, idc_i\rangle$ to its corresponding dataset $\mathcal{H}_i$ (lines 2 to 4 of Algorithm 2). Taking $\mathcal{H}_{\langle\langle x,y\rangle, \langle idc_1, idc_2\rangle\rangle}$ as an example, after the above operations, we get two data sets:

$$\mathcal{H}_{\langle x,y,idc_1\rangle} = \{\langle 2,1,0\rangle, \langle 1,1,1\rangle, \langle 1,0,0\rangle, \langle 0,2,2\rangle, \langle 0,1,1\rangle\}$$
$$\mathcal{H}_{\langle x,y,idc_2\rangle} = \{\langle 2,1,2\rangle, \langle 1,1,1\rangle, \langle 1,0,1\rangle, \langle 0,2,0\rangle, \langle 0,1,0\rangle\}$$

Next, we apply SimpleBoundLearn (or ConjunctiveBoundLearn) on $\mathcal{H}_{\langle x,y,idc_1\rangle}$ and $\mathcal{H}_{\langle x,y,idc_2\rangle}$ respectively, and learn two expressions $m_1 = y$ and $m_2 = x$. Sequentially combining them together, we get a lexicographic loop bound candidate $\mathbf{m}(x, y) = \langle y, x\rangle$.

4.4 Discussion

There exist other techniques for learning simple loop bounds, e.g. [31]. Compared to [31], our simple bound learning approach adds a new parameter *mCost* and requires the bound candidates to tend in a natural form.

The paper [41] learns a set of affine functions through a series of simple and random selections for data examples. After that, it arranges these affine functions into a bound template with conjunctive or lexicographic forms directly. Hence, it cannot support compound form bound learning (e.g., lexicographic bound with conjunctive bound in each dimension). In contrast, our conjunctive bound learning considers more global and local features among the dataset, including the distance information (cluster) and the outlier examples (slack set). Moreover, our lexicographic bound learning analyzes the executing trace information. With the help of

Algorithm 3: Data-Driven Termination Analysis

```
   input  : A program $P$
   output : TERMINATES or UNKNOWN
1  $\mathbf{m}, \mathcal{H} \leftarrow \mathbf{0}, \emptyset$
2  repeat
3      $\Phi \leftarrow$ VerificationTask($P, \mathbf{m}$)
4      cex $\leftarrow$ Check($\Phi$)
5      if cex $= \emptyset$ then
6          return TERMINATES
7      else
8          $\mathcal{H} \leftarrow \mathcal{H} \cup$ GetDataFromCEX($P$, cex)
9          $\mathbf{m} \leftarrow$ BoundLearn($\mathcal{H}$)
10 until TIMEOUT
11 return UNKNOWN
```

the additional information, our approach becomes more effective and efficient.

5 DATA-DRIVEN TERMINATION ANALYSIS

A validated loop bound proves the program's termination. This section presents our loop bound-based termination analysis approach.

5.1 Overall Algorithm

Algorithm 3 presents our data-driven termination analysis algorithm. After initialization (line 1), the algorithm iteratively updates the loop bound $\mathbf{m}$ and the dataset $\mathcal{H}$ (lines 2-9). Each iteration generates a validation task Φ from the input program P and the current loop bound $\mathbf{m}$ and then sends the validation task to a checker. If Φ passes the checking, the current loop bound is validated, the algorithm returns TERMINATES (line 6). Otherwise, the checker returns a counterexamples cex, the algorithm then proceeds to enrich the dataset $\mathcal{H}$ with the data examples obtained from cex (line 8) and learn a new loop bound $\mathbf{m}$ from this enriched dataset (line 9). The above procedure repeats until either a validated loop bound is found or TIMEOUT is reached; for the latter case, the algorithm returns UNKNOWN.

5.2 Validation Task

With a program P and a loop bound candidate $\mathbf{m}$, the *validation task* is a transformation [14, 41] to insert some statements related to the validation of $\mathbf{m}$ into the program P. Suppose the program P is in the following form:

$$\textbf{assume}(Pre(\mathbf{x}));\ \textbf{while}(Guard(\mathbf{x}))\ \{Trans(\mathbf{x}, \mathbf{x}');\} \quad (4)$$

where $Pre(\mathbf{x})$ is the precondition representing the codes before the loop, $Guard(\mathbf{x})$ is the loop guard, and $Trans(\mathbf{x}, \mathbf{x}')$ represents the loop body that updates the program variable from $\mathbf{x}$ to $\mathbf{x}'$.

Then, the validation task transforms the program (4) with loop bound $\mathbf{m}$ into the following form:

$$\begin{aligned}&\textbf{assume}(Pre(\mathbf{x}));\ \textbf{assume}(\mathbf{i} \succcurlyeq \mathbf{m}(\mathbf{x})); \quad (5)\\ &\quad \textbf{while}(Guard(\mathbf{x}))\ \{\textbf{assert}(\mathbf{i} \succ \bot);\ \texttt{Dec}(\mathbf{i}, \mathbf{i}');\ Trans(\mathbf{x}, \mathbf{x}');\}\end{aligned}$$

where $\texttt{Dec}(\mathbf{i}, \mathbf{i}')$ is a decreasing function on the well-ordered set $\mathcal{W}$ which updates the *validation counter* $\mathbf{i}$ into $\mathbf{i}'$.

Validate Simple & Conjunctive Bound Candidates. In this case, both the bound candidate $\mathbf{m}$ and the validation counter $\mathbf{i}$ are 1-dimensional scalars and belong to $\mathbb{N}$. Then, $\mathbf{i} \succcurlyeq \mathbf{m}(\mathbf{x})$ becomes $i \geq m(\mathbf{x})$, $\mathbf{i} \succ \bot$ becomes $i > 0$, and the decreasing function $\texttt{Dec}(\mathbf{i}, \mathbf{i}')$ becomes $i' = i - 1$. Figure 2 shows an example of the validation task transformed from the program in Figure 1, where m(x) can be instantiated using the learned loop bound function.

Validate Lexicographic Bound Candidates. For an n-dimensional lexicographic loop bound $\mathbf{m}$, the well-ordered set $\mathcal{W}$ is a set of n-dimensional vectors, of whom each dimension belongs to $\mathbb{N}$. The bottom element $\bot$ is the zero-valued vector $\mathbf{0}$. For two vectors $\mathbf{v}, \mathbf{v}' \in \mathcal{W}$, the well-founded relation $\succ_n$ over them is defined as

$$\mathbf{v} \succ_n \mathbf{v}' \triangleq \exists i > 0.\ \mathbf{v}[i] > \mathbf{v}'[i] \wedge \forall j \in (i, n].\ \mathbf{v}[j] = \mathbf{v}'[j].$$

For example, we have $\langle 1, 2\rangle \succ_2 \langle 2, 1\rangle$ and $\langle 2, 1\rangle \succ_2 \langle 1, 1\rangle$ in the 2-dimensional $\mathcal{W}$.

In the transformation, we employ n variables from i1 to in to represent the n-dimensional validation counter vector $\mathbf{i}$. The decreasing function $\texttt{Dec}(\mathbf{i})$ can be described as follows. We check the value $\mathbf{i}[j]$ for $j = 1$ to n until we find j^* such that $\mathbf{i}[j^*] > 0$. Then, we decrease $\mathbf{i}[j^*]$ by 1. For all $1 \leq j^+ < j^*$, we reset $\mathbf{i}[j^+]$ to a value greater than or equal to $\mathbf{m}[j^+]$. After executing these steps, the value of $\mathbf{i}$ becomes $\mathbf{i}'$. We can easily prove that $\mathbf{i} \succ \mathbf{i}'$. For example, the program in Figure 5 is the validation task transformed from the program in Figure 4 with a 2-dimensional *LexLB*, where lines 5 to 9 encode the 2-dimensional decreasing function $\texttt{Dec}(\mathbf{i})$.

5.3 Safety Checker

Figure 6 demonstrates our loop bound learning and validation framework. In line 4 of algorithm 3, after the bound candidate is learned, a validation task will be created and delivered to the safety checker for bound validation.

Finding an appropriate *loop invariant* is a basic way for the safety checker to prove the problem with loops safe. In our work, we use a data-driven loop invariant learning approach [16]. Similar to bound learning, most of the data-driven approaches for loop invariant synthesis [16, 22, 32, 42] also use program states at the loop header as their data examples. There are two sets of data examples for invariant learning $\mathcal{S}^+$ and $\mathcal{S}^-$. Consider the program (5). $\mathcal{S}^+$ includes those program states which are reachable from the loop precondition $Pre(\mathbf{x}) \wedge \mathbf{i} \succcurlyeq \mathbf{m}(\mathbf{x})$, and $\mathcal{S}^-$ includes those program states from which there exists a reachable trace leading to a failed assertion $\neg(\mathbf{i} \succ \bot)$.

The safety property requires no state in both $\mathcal{S}^+$ and $\mathcal{S}^-$. Otherwise, let $\mathbf{s} \in \mathcal{S}^+ \cap \mathcal{S}^-$. According to the definition, we can easily construct an error trace from the loop precondition to the failed safety property assertion through the state $\mathbf{s}$ and report a bound validation error (the gray edge from bound validation to bound learning in Figure 6). When $\mathcal{S}^+ \cap \mathcal{S}^- = \emptyset$, we use [16] with $\mathcal{S}^+$ and $\mathcal{S}^-$ to learn a loop invariant hypothesis. If the safety checking passes with the hypothesis, the termination will be reported (the right-most outedge from bound validation in Figure 6). In the other case, $\mathcal{S}^+$ and $\mathcal{S}^-$ will be extended to refine the invariant hypothesis in the next round (the gray cycle between invariant examples and the safety checker in bound validation in Figure 6).

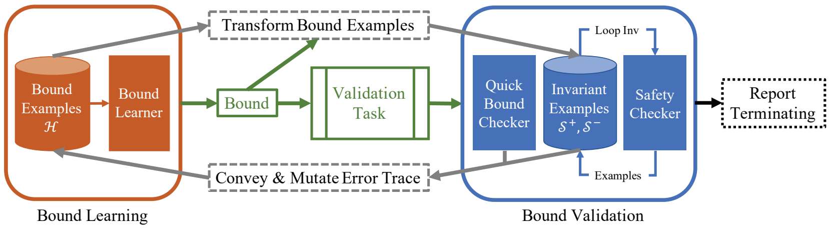

Figure 6: Bound learning and validation framework

Algorithm 4: Check(Φ)

```
input  : A validation task $\Phi$
output : a counterexample cex or $\emptyset$
1 $k^* \leftarrow$ GetMaxUnroll($\Phi$)           // Quick Checking
2 for $k \leftarrow 0$ to $k^*$ do
3     if FeasibilityCheck($\Phi, k$) = UNSAT then
4         return $\emptyset$
5     cex $\leftarrow$ QuickBoundCheck($\Phi, k$)
6     if cex $\neq \emptyset$ then
7         return cex
8 return SafetyCheck($\Phi$) // Check by Safety Verifier
```

5.4 Quick Bound Checking

The check(Φ) procedure in Algorithm 3 is to check the correctness of the safety property in Φ. In case of incorrectness, it needs to return a counterexample that refutes the current bound candidate.

Keep in mind that it usually requires a considerable number of iterations for learning a correct loop bound. In all but the last iteration, the input loop bound fails the validation, and the check(Φ) procedure actually acts like a counterexample generator in these iterations. Since that falsification is always cheaper than verification, it is thus worthwhile to apply a quick falsification check before the check(Φ) procedure. To this end, we propose a bounded model checking[4, 10] (BMC)-based *quick bound checking* to enhance the check(Φ) procedure.

The enhanced check(Φ) procedure is presented in Algorithm 4. For a given validation task Φ, it first performs a BMC-style loop unrolling from 0 to k^* and makes a series of incremental SMT queries as follows:

$$Pre(\mathbf{x}_0) \wedge \mathbf{i}_0 \succcurlyeq \mathbf{m}(\mathbf{x}_0) \wedge \bigwedge_{j=0}^{k-1} (Guard(\mathbf{x}_j) \wedge Trans(\mathbf{x}_j, \mathbf{x}_{j+1}) \wedge Dec(\mathbf{i}_{j+1}, \mathbf{i}_j) \wedge Guard(\mathbf{x}_k) \quad (6)$$

$$Pre(\mathbf{x}_0) \wedge \mathbf{i}_0 \succcurlyeq \mathbf{m}(\mathbf{x}_0) \wedge \bigwedge_{j=0}^{k-1} (Guard(\mathbf{x}_j) \wedge Trans(\mathbf{x}_j, \mathbf{x}_{j+1}) \wedge Dec(\mathbf{i}_{j+1}, \mathbf{i}_j) \wedge Guard(\mathbf{x}_k) \wedge \neg \mathbf{i}_k \succ \bot \quad (7)$$

The query (6) asks whether the loop body is still feasible after k times of loop iterations. If this query is unsatisfiable, the program terminates with k times of loop iterations. As a result, the algorithm just returns an empty set. Otherwise, we make the query (7), which asks whether the current loop bound $\mathbf{m}(\mathbf{x})$ is violated after k times of loop iterations. If this query is satisfiable, there exists a k-steps counterexample trace where the number of loop iterations exceeds the initial value of $\mathbf{m}(\mathbf{x}_0)$. Then, we return this counterexample cex. If no counterexample is found in the above quick checking procedure, we invoke a safety verifier to exhaustively check Φ.

In our motivation example, the loop bound candidates m_1, m_2 and m_3 can be easily refuted by quick bound checking with $k^* \leq 1$.

5.5 Two-way Example Sharing

Section 5.3 shows that both bound learning examples and invariant learning examples are the program states at the loop header. The similarity inspires us to investigate their potential connections.

In our termination analysis framework, we design a two-way data sharing between bound and invariant examples, i.e., the gray edges in Figure 6. Firstly, the bound examples can be transformed into invariant examples in the bound validation. Secondly, the error traces provided by the bound validation to refute the bound candidate can be conveyed and mutated to generate more bound learning examples.

Transform Bound Examples. Recall that the examples in dataset $\mathcal{H}$ for the bound learning are in the form of a pair $\langle \mathbf{x}, \mathbf{idc} \rangle$ where $\mathbf{x}$ is a program state at loop header, and **idc** is the remaining iteration number for $\mathbf{x}$ in an executing trace.

The main differences between bound learning examples and invariant learning examples mainly lie in two aspects:

(1) Comparing to the original program (4), the program states in validation task (5) have a particular decreasing counter $\mathbf{i}$;
(2) Learning invariant needs two sets, i.e., $\mathcal{S}^+$ and $\mathcal{S}^-$. However, Bound learning only uses one set, i.e., $\mathcal{H}$.

In our approach, attached with a concrete value of $\mathbf{i}$, each bound learning example $\langle \mathbf{x}, \mathbf{idc} \rangle$ can be transformed into an invariant learning example $\mathbf{s}^+ \in \mathcal{S}^+$ and an invariant learning example $\mathbf{s}^- \in \mathcal{S}^-$.

Let us explain our idea with a 1-dimensional bound m for easy understanding. Suppose the current bound candidate is $m(\mathbf{x})$ and bound learning example $\langle \mathbf{x}, idc \rangle$ is from an actual executing trace:

$$\langle \mathbf{x}_0, idc_0 \rangle \to \langle \mathbf{x}_1, idc_1 \rangle \to \ldots \to \langle \mathbf{x}_k, idc_k \rangle$$

where the iteration down counter $idc_t = k - t + 1$ for all $t \in [0, k]$.

We can construct a program state example $\mathbf{s}_0^+$ for program (5) with $\mathbf{x}_0$ and a validation counter $i_0 = m(\mathbf{x}_0)$. Obviously, because the program state $\mathbf{x}_0$ is reachable from the precondition in program (4), the program state $\mathbf{s}_0^+$ is reachable from the precondition in program

(5), i.e., $\mathbf{s}_0^+ \in \mathcal{S}^+$. Furthermore, we can also prove that the program state $\mathbf{s}_t^+$ constructed with $\mathbf{x}_t$ and $i_t = m(\mathbf{x_0}) + t - k$ for all $t \in [0, k]$ are also reachable from the precondition, i.e., $\mathbf{s}_t^+ \in \mathcal{S}^+$.

Similarly, we can construct a program state example $\mathbf{s}_k^-$ for program (5) with $\mathbf{x}_k$ and a validation counter $i_k = 0$. Obviously, $\mathbf{s}_k^-$ violates the safety property assertion in program (5), i.e., $\mathbf{s}_k^- \in \mathcal{S}^-$. Furthermore, because there exists a trace from $\mathbf{x}_0$ to $\mathbf{x}_k$ in program (4), we can also prove that from the program state $\mathbf{s}_t^-$ constructed with $\mathbf{x}_t$ and $i_t = idc_t - 1$ for all $t \in [0, k]$ there also exists an executing trace to the falsified assertion $\mathbf{s}_k^-$ in program (5), i.e., $\mathbf{s}_t^- \in \mathcal{S}^-$.

Convey & Mutate Error Traces. An error trace will be reported when the safety checker fails in the bound validation. We can use the length from $\mathbf{x}$ to the falsified assertion in the reported error trace to calculate **idc** and convey $\langle \mathbf{x}, \mathbf{idc} \rangle$ to the bound examples set.

We believe that the program states near the error trace are very likely to lead to the same safety property violations with similar causes. So, in order to fully explore the error causes, besides conveying the reported error trace, we also make some mutation tests near the error trace to generate more bound learning examples. In mutation tests, we slightly perturb a small number of variables in the program according to their assignments in the error trace and observe the loop bound of its execution.

6 EVALUATION

We implement a prototype tool called ddlTerm[2], which uses a data-driven invariant learner ICE-DT [15, 16] as the backend safety checker. To evaluate the effectiveness and efficiency of ddlTerm, we compare it with the state-of-the-art termination analysis tools:

- FreqTerm [14] synthesizes loop bounds by a SyGuS method and validates them by CHC solvers. Three CHC solvers, i.e., FreqHorn, μZ, and Spacer are supported in FreqTerm.
- UAutomizer [9, 20] proves termination by decomposing programs into modules, and synthesizing its termination arguments for each module.
- AProVE [17, 18, 40] proves termination by transforming programs into *term rewrite systems.*
- MuVal [24] learns ranking functions by decision trees.

The benchmark programs are collected from [14]. There are a total of 170 terminating programs in [14]. We take *all of them* in our experiments. All experiments are conducted on a computer with Intel(R) Core (TM) i7-9700 CPU (3.00 GHz) and 32 GB memory, running Ubuntu 18.04 platform. The timeout limit for each benchmark is set to 120 seconds.

6.1 Efficiency Evaluation

This experiment compares our tool ddlTerm with FreqTerm, UAutomizer, MuVal, and AProVE. Note that FreqTerm can be configured with three different backend solvers, i.e., FreqHorn, μZ and Spacer. All three configurations of FreqTerm will be experimented with.

[2]The tool is available at: https://doi.org/10.5281/zenodo.5442280.

Overall Result. The overall result is shown in Table 2. In total 170 benchmarks, ddlTerm solved 136 of them. ddlTerm solves **35-48%** more benchmarks than FreqTerm, **24%** more benchmarks than UAutomizer, **13%** more benchmarks than MuVal, and **27%** more benchmarks than AProVE. We also calculated the average time spent on solving each benchmark. Our ddlTerm needs 5.43s, while FreqTerm needs 9.20s on average (**41%** improved), UAutomizer needs 17.29s (**69%** improved), MuVal needs 9.05s (**40%** improved), and AProVE needs 23.56s (**77%** improved).

We also present more detailed numbers of solved cases in Table 2 for the baselines. ddlTerm and baselines have their own advantages on different benchmarks because of the different technical routes. However, generally speaking, there are many cases that ddlTerm can solve while the baseline cannot, and a small number of cases that the baseline can solve while ddlTerm cannot.

Overall Comparison. We present the curve graph of cumulative running time for solved benchmarks of all tools in Figure 8. In this figure, it is easy to find that our ddlTerm has both stronger solving ability and relatively higher efficiency compared with FreqTerm, UAutomizer, MuVal, and AProVE.

Separate Comparison. For each baseline, we present a scatter plot in Figure 7. Each point represents the running time for one specific benchmark of ddlTerm (x-axis) and the baseline (y-axis). If the point reaches the top or right bound in the figure, it means the baseline or ddlTerm fails in solving this benchmark. In these figures, the more the points are close to the top-left corner, the better our tool is. As we can see, ddlTerm significantly outperforms FreqTerm, UAutomizer, MuVal, and AProVE on the number of solved benchmarks and the distributions of running time.

6.2 Effectiveness Evaluation

We also conduct several other experiments to evidence the effectiveness of our approaches.

Effectiveness of Bound Learning Algorithms. In this part, we evaluate the solving ability gain, which benefits from our bound learning methods. Figure 9 shows the number of additionally solved benchmarks when a new bound learning algorithm is applied. In this figure, the *yellow part with the northwest pattern* represents the cumulative number of solved benchmarks only using the *simple loop bound* learning (no lexicographic and no conjunctive). The *purple part with the northeast pattern*, the *red part with the vertical pattern*, and the *green part with the horizontal pattern* respectively represent the cumulative numbers of solved benchmarks when the *conjunctive loop bound* learning (no lexicographic but conjunctive), the *lexicographic simple loop bound* learning (lexicographic but no conjunctive) and the *lexicographic conjunctive loop bound* learning (lexicographic and conjunctive) are used. In total, the simple bound learning method solved 92 benchmarks, and the conjunctive bound learning method, the lexicographic simple bound learning method, and the lexicographic conjunctive bound learning method respectively solved 27%, 9%, and 6% more benchmarks than their former.

Effectiveness of Quick Bound Checking. Recall that both the *quick bound checking* and the *safety verifier* can provide a counterexample

Table 2: General result for efficiency evaluation

Tool	ddlTerm	FreqTerm			UAutomizer	MuVal	AProVE
		FreqHorn	μZ	Spacer			
#Solved.	**136**	92	93	101	110	120	107
#Both Solved.	-	88	86	89	91	104	93
#ddlTerm Only.	-	48	50	47	45	32	43
#Baseline Only.	-	4	7	12	19	16	14
Time(s)	**4818**	10818	9616	9058	9102	7086	10080
Time on Solved(s)	**738**	1459	376	778	1902	1086	2520
Avg. T. on Solved(s)	**5.43**	15.85	4.04	7.70	17.29	9.05	23.56
		Average: 9.20					

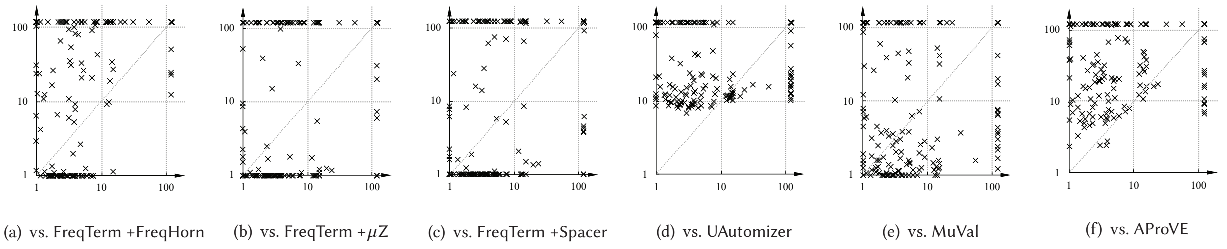
(a) vs. FreqTerm +FreqHorn (b) vs. FreqTerm +μZ (c) vs. FreqTerm +Spacer (d) vs. UAutomizer (e) vs. MuVal (f) vs. AProVE

Figure 7: Separate comparison on running time (x-axis represents ddlTerm and y-axis represents each baseline)

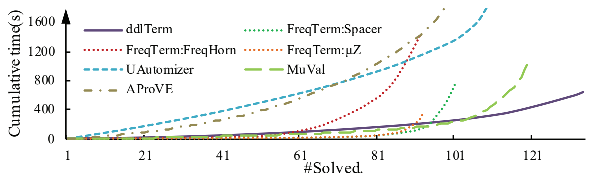

Figure 8: Cumulative time for solved benchmarks

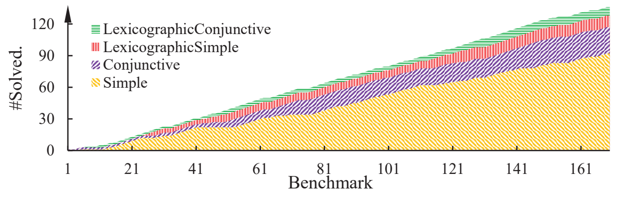

Figure 9: Comparison on different bound learning strategies

to refine the incorrect loop bound candidate. We count the number of these refinement rounds during the termination analysis. We also track the counterexample provider of each refinement, i.e., either the quick bound checker or the safety checker.

The stacked histogram in Figure 10 shows the experimental results. To make the histogram clearer, we use two sub-figures with different scales for the number of refinements (y-axis). The *red part* in each bar represents the number of refinements advised by the quick bound checking, and the *blue part* represents the number of refinements advised by the safety verifier. The *filled bars* represent the successfully proved benchmarks, and the *hollow bars* represent benchmarks that are not proved. In this figure, it is obvious that our quick bound checking is generally working during the analysis. It also takes a significant proportion in the refinements for loop bound candidates.

Next, we investigate the solving ability enhancement and the time reduction benefiting from the quick bound checking. The quick bound checking is *disabled* in the baseline. The results are shown in Table 3. In this table, column "#Sol." represents the numbers of benchmarks solved by ddlTerm *with* or *without* quick bound checking (Q.B.C.). With quick bound checking, ddlTerm solves 29 more benchmarks. Notice that the other columns on the right represent the result on 107 benchmarks both strategies solved. Column "T." represents the total running time. Columns "T.I" and "T.Q" represent the time used by the safety verifier and the quick bound checking, respectively. Column "#R" represents the numbers of total numbers of the loop bound refinements. Columns "#R.I" and "#R.Q" represent the numbers of refinements advised by the safety verifier and the quick bound checking, respectively. With quick bound checking, ddlTerm not only solves 29 more benchmarks but also reduces 186s spent by safety verifier (**46%**) and 64s in total (**9%**). Although it takes more iterations on loop bound refinements with quick bound checking, according to the improvement of the overall running time, we can still conclude that quick bound checking is necessary and makes our approach more efficient.

Effectiveness of Two-way Data Sharing. With our two-way data sharing, the safety checker obtains the transformed data examples

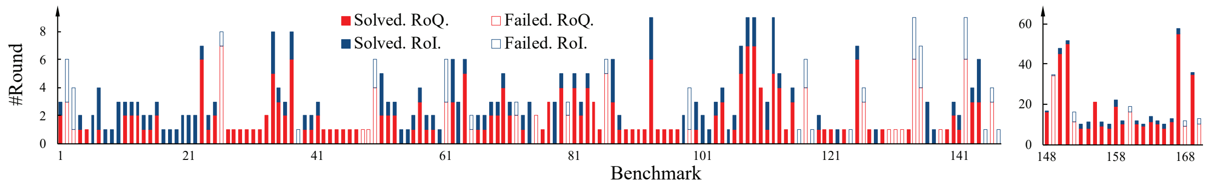

Figure 10: The distribution of different bound refinements

Table 3: Comparative results of quick bound checking

Strategy	#Sol.	On 107 benchmarks both solved.					
		T.(s)	T.I(s)	T.Q(s)	#R.	#R.I	#R.Q
With Q.B.C.	**136**	**631**	**219**	79	355	**112**	243
Without Q.B.C.	107	695	405	-	210	210	-

from the bound learning, and the bound learner also benefits from the counterexamples generated by the safety checker. In this experiment, we disable the data sharing in each direction and evaluate our approach.

Table 4: Comparative results of two-way data sharing

Strategy	#Solved.	Time(s)	T. on Sol.(s)	Avg. T. on Sol.(s)
Enable Both	**136**	**4818**	**738**	**5.43**
Disable T.B.E.	136 (0)	4882	802	5.90 (8.0%)
Disable M.E.T.	135 (-1)	5091	891	6.60 (17.7%)

Results are shown in Table 4. The first row, "Enable Both", represents the strategy where both two directions of data sharing are both enabled. The second row, "Disable T.B.E.", means we disable the transformation of bound examples to invariant examples. "Disable T.B.E." is 8% slower than "Enable Both" in average time. The third row, "Disable M.E.T.", means we disable the mutation test on error trace. Actually, to make the interactions between bound learning and validation go on, we cannot fully prevent the data sharing from the safety checker to the bound learner, i.e., the conveyance of error traces still exists. "Disable M.E.T." solves one benchmark less and is 17.7% slower in average time than "Enable Both" . We can conclude that our two-way data sharing mechanism between bound learning and validation makes our approach more efficient.

7 RELATED WORK

Proving Termination. Termination analysis has always been a research hotspot in the field of program verification, and there are amount of existing technologies. Most of them [3, 6, 19, 25, 28, 33, 35] rely on constraint solvers to synthesize the termination arguments, e.g., a ranking function together with its support invariant. With these technologies, lots of tools emerged, e.g., Terminator (together with its successor T2) [7, 11, 12], Armc[34], Tan[23], HipTNT+ [27], Ultimate Automizer [20] and so on.

Differing from the technologies above, there are some "*guess-and-check*"-style methods [14, 26, 31, 41]. They employ a lightweight approach to obtain the likely termination arguments and employ an off-the-shelf checker to validate them. DynamiTe [26] tries to get a ranking function candidate from dynamic execution traces and employs a program verifier to check it. Its method is different from ours, and it focuses on non-linear programs. FreqTerm [14] uses a syntax-guided approach to guess a loop bound and checks the candidate by a CHC solver. [41] learns different forms of loop bounds by combining several affine expressions inferred from the examples provided by the constraint solver. TpT [31] employs quadratic programming to infer a loop bound from testing data, which is closest to our approaches. However, TpT only learns a simple loop bound. Its applicability is thus limited.

Data-Driven Methods in Verification. Data-Driven methods have received more and more attention in recent years. In the field of verification, several applications have been appeared, particularly in loop invariant synthesis [16, 29, 37, 38, 42, 43] and termination proving [24, 26, 31, 41]. The ways these technologies deal with the data are quite different. Among these technologies, [16, 24, 42] use adapted decision trees to process the data. [29, 37] use SVM for data classification. [38, 43] use natural networks to generate logical expressions. [31] uses quadratic programming, [41] uses linear interpolation, and [26] uses constraint solving. In our approaches, we use the clustering algorithm to group the dataset and then employ convex optimization to generate the candidate loop bounds. We also employ the optimization algorithm to find a better combination of the bound candidates.

8 CONCLUSION

We develop a data-driven loop bound learning approach for termination analysis. We propose a series of data-driven loop bound learning algorithms, i.e., simple loop bound learning, conjunctive loop bound learning, and lexicographic loop bound learning. With a combination of these learning algorithms, our approach is able to prove the termination of complicated programs with non-linear loop bounds. We also propose a quick bound checking method to efficiently refute the incorrect loop bound by a counterexample. We also design two-way data sharing to bridge the bound learning and validation. We conduct extensive experiments to evaluate the efficiency and effectiveness of our approach. Our tool significantly outperforms the state-of-the-art termination analysis tools.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002), and the National Key Research and Development Program of China (No. 2018YFB1308601).

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.

[2] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod record* 28, 2 (1999), 49–60.

[3] Amir M Ben-Amram and Samir Genaim. 2013. On the linear ranking problem for integer linear-constraint loops. *ACM SIGPLAN Notices* 48, 1 (2013), 51–62.

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. (2003).

[5] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

[6] Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2005. Termination analysis of integer linear loops. In *International Conference on Concurrency Theory*. Springer, 488–502.

[7] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better termination proving through cooperation. In *International Conference on Computer Aided Verification*. Springer, 413–429.

[8] Alberto Caprara, Paolo Toth, and Matteo Fischetti. 2000. Algorithms for the set covering problem. *Annals of Operations Research* 98, 1 (2000), 353–371.

[9] Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced automata-based algorithms for program termination checking. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 135–150.

[10] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design* 19, 1 (2001), 7–34.

[11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. *ACM Sigplan Notices* 41, 6 (2006), 415–426.

[12] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. lexicographic termination proving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 47–61.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

[14] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-guided termination analysis. In *International Conference on Computer Aided Verification*. Springer, 124–143.

[15] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*. Springer, 69–87.

[16] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.

[17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. 2017. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning* 58, 1 (2017), 3–31.

[18] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. 2014. Proving termination of programs automatically with AProVE. In *International Joint Conference on Automated Reasoning*. Springer, 184–191.

[19] William R Harris, Akash Lal, Aditya V Nori, and Sriram K Rajamani. 2010. Alternation for termination. In *International Static Analysis Symposium*. Springer, 304–319.

[20] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination analysis by learning terminating programs. In *International Conference on Computer Aided Verification*. Springer, 797–813.

[21] Dieter Kraft et al. 1988. A software package for sequential quadratic programming. (1988).

[22] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. 2015. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725* (2015).

[23] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M Wintersteiger. 2010. Termination analysis with compositional transition invariants. In *International Conference on Computer Aided Verification*. Springer, 89–103.

[24] Satoshi Kura, Hiroshi Unno, and Ichiro Hasuo. 2021. Decision Tree Learning in CEGIS-Based Termination Analysis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 75–98. https://doi.org/10.1007/978-3-030-81688-9_4

[25] 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*. IEEE, 218–225.

[26] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[27] 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*. 489–498.

[28] Jan Leike and Matthias Heizmann. 2014. Ranking templates for linear loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 172–186.

[29] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation anc refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 782–792.

[30] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA, 281–297.

[31] Aditya V Nori and Rahul Sharma. 2013. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 246–256.

[32] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices* 51, 6 (2016), 42–56.

[33] Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 239–251.

[34] Andreas Podelski and Andrey Rybalchenko. 2007. ARMC: the logical choice for software model checking with abstraction refinement. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 245–259.

[35] Andreas Podelski and Andrey Rybalchenko. 2011. Transition invariants and transition predicate abstraction for program termination. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 3–10.

[36] Michael JD Powell. 1994. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*. Springer, 51–67.

[37] Rahul Sharma, Aditya V Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *International Conference on Computer Aided Verification*. Springer, 71–87.

[38] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. In *Neural Information Processing Systems*.

[39] Hugo Steinhaus et al. 1956. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci* 1, 804 (1956), 801.

[40] Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. 2015. AProVE: termination and memory safety of C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 417–419.

[41] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing ranking functions from bits and pieces. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 54–70.

[42] 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.

[43] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.