Loopster++: Termination Analysis for Multi-path Linear Loop

Hui Jin[1], Weimin Ge[1], Yao Zhang[1], Xiaohong Li[1(✉)], and Zhidong Deng[2]

[1] College of Intelligence and Computing, Tianjin University, Tianjin, China
{jinh,gewm,xiaohongli}@tju.edu.cn
[2] State Grid Customer Service Center, Tianjin, China

Abstract. Loop structure is widely adopted in many applications, *e.g.* collaborative applications, social network applications, and edge computing. And the termination of the loop is of great significance to the correctness of the program. Most of the previous relative studies focused on determining the termination of a loop program by synthesizing the ranking functions, but not every ranking function can be synthesized. Although a class of linear loop program termination has been proven to be decidable, it is always difficult to analyze the termination of a multi-path linear loop. Xie et al. [20] presented Loopster to quickly check the termination of the multi-path loop program by analyzing the termination of each path and the dependency between paths. But it relies on the monotonicity of variables which is very complicated to check when the variables increase.

To this end, we extend Loopster, named Loopster++, to analyze the termination of multi-path linear loops. In Loopster++, 1) we convert the iterable path into a single path linear loop to analyze its termination. 2) We also propose a novel method to analyze the dependency between linear loop paths. 3) For the cycle constituted by alternate execution between paths, we classify all cycles and give the termination method of the corresponding category cycle. We finally evaluate Loopster++ by analyzing the termination of the benchmarks from the competition on software verification and compare it with the state-of-the-art tools. The empirical results demonstrate the superiority of Loopster++ by achieving high accuracy of 83% in the shortest time.

Keywords: Termination analysis · Multi-path linear loop · Path dependency automaton

1 Introduction

The termination analysis of the program is one of the most important parts of the program verification, and it is of great significance to ensure the correctness of the program. Furthermore, non-termination will cause a variety of program bugs, even incurring denial-of-service attacks [2], and be hardly notified [12]. Therefore, it is imperative to determine the termination of the programs.

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 406, pp. 479–497, 2021.
https://doi.org/10.1007/978-3-030-92635-9_28

The research on program termination analysis has received a lot of advances. The general approaches are to synthesize the ranking functions [3,7,9,14,17]. Colon and Sipma [7] proposed a method to synthesize the ranking functions based on polyhedral cones and systems of linear constraints. Podelski and Rybalchenko [17] proposed the synthesis of linear ranking functions for linear loop programs except for nested loops. Ben-Amram and Genaim [3] synthesize the ranking functions by integrating polyhedra theory to prove the termination of a loop program which has linear constraints. Leike and Heizmann [14] proposed a ranking template, which covers all methods based on constraint-based synthetic ranking functions. However, having a ranking function is a sufficient condition for termination, which is no longer effective if cannot be founded. Therefore, there are other techniques to analyze the decidable class of loops [5,16,18]. Tiwari [18] showed that the termination of a linear loop of $while(Bx > b)\{x = Ax + c\}$ form is decidable range over $\mathbb{R}$. Braverman [5] generalized the work of Tiwari and proved that the homogeneous form is decidable over integers. If the update matrix A of a simple linear loop program can be diagonalized, Ouaknine et al. [16] proposed how to decide its termination. Note that, these techniques only consider the single path, however, a loop program is normally multi-path in practice.

To analyse loop programs, Xie et al. [19] presented a loop summary framework, namely Path Dependency Automaton (PDA), which summarizes path-sensitive loop on interesting variables. They extracts the properties of each path, and then summarizes the properties of the overall loop based on the dependencies between paths. However, In their approach, all the related variables need to be inductive when building PDA. To efficiently determine the loop termination, Xie et al. further proposed Loopster [20] based on PDA by analyzing the monotonicity of variables, which is not only limited to the inductive variables. But for linear loop programs, the monotonicity of variables is hardly detected.

In this paper, we extend the Loopster to Loopster++ based on the theory of termination of linear loop proposed by Tiwari [18]. Specifically, Loopster++ follows the theory in [18] to analyze the termination of each iterable path so that it is capable of extending the PDA to support linear expression in the loop program. At the same time, we propose a novel method to analyze the dependency between the paths in the multi-path linear loop. Finally, we also present a method to analyze the termination of the cycle generated in this linear loop. To demonstrate the effectiveness of Loopster++, we apply it to analyze the benchmarks from the competition on software verification [1] and we compare it with the top-three tools, i.e., Ultimate Automizer, CPAchecker, and 2LS. The result shows that Loopster++ can correctly handle 100 loop programs in a total of 120 loop programs. Meanwhile, Loopster++ outperforms other tools on efficiency, which is at least 5x faster than the other three tools.

In summary, this paper makes the following contributions:

– We follow the theory of linear loop termination to extend Loopster and propose Loopster++ so that it is capable of supporting multi-path linear loops.

– We extend the algorithm proposed by Tiwari [18] so that the termination of the loop with precondition can be well determined.
– In order to deal with the cycle formed by strong connected components between paths, we make a classification of the cycle derived from Loopster++ and propose the corresponding method to determine the termination of the cycle.

2 Preliminaries

In this section, we define the scope of our work, some professional terms in PDA proposed by Xie et al. [19], the structure of Loopster [20] and the determination algorithm of the termination of linear loop programs.

2.1 Scope of Our Work

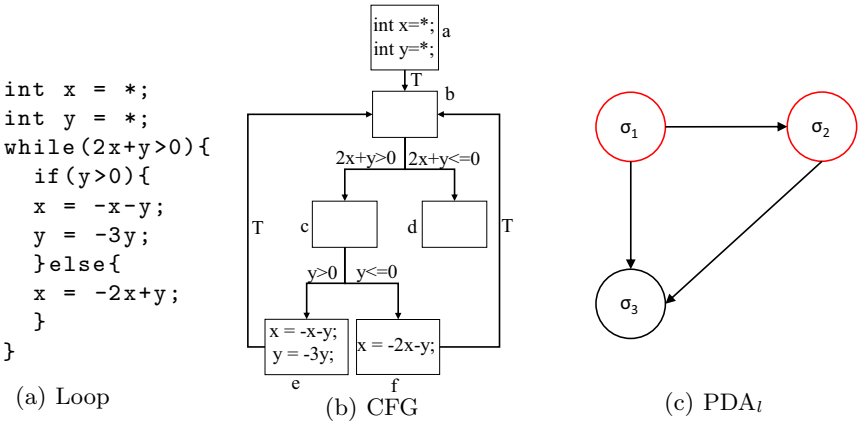
We focus on the termination analysis of multi-path linear loops in this paper. Let $X = \{x_1, x_2, ..., x_n\}$ be a finite set of variables ranging over $\mathbb{R}$, and $f(x_1, x_2, ..., x_n)$ be a multivariate linear polynomial. All atomic operations on a loop are in the form of $f(x_1, x_2, ..., x_n) \sim b$, $b \in \mathbb{R}$ and $\sim \in \{<, \leq, >, \geq, =\}$ (= represents the assignment operation). Limited by the linear loop termination analysis algorithm, other operations are not considered in this paper.

2.2 Path Dependency Automaton (PDA)

A control flow graph (CFG) of a loop is a tuple $\mathcal{G} = \{V, E, v_s, V_h, V_e, \iota\}$. Where V is a set of basic blocks, E is a set of directed edges connecting two basic blocks, v_s is the start node of the CFG, executed before entering the loop, V_h and V_e are the set of header blocks and exit blocks of the loop, $\iota(e)$ is the branch condition of the edge $e \in E$.

Example 1. Figure 1(b) shows a CFG of the unnested loop in Fig. 1(a), where $V = \{a, b, c, d, e, f\}$, $E = \{(a, b), (b, c), (b, d), (c, e), (c, f), (e, b), (e, f)\}$, $v_s = a$, $V_h = \{b\}$, $V_e = \{d\}$ and $\iota((b, c)) = 2x + y > 0$. Since there is only one loop, there is only one header block. In Fig. 2(b), since the nested loop has two loops, the head blocks of this loop are b and c.

Given a control flow graph $\mathcal{G} = \{V, E, v_s, V_h, V_e, \iota\}$, the loop path σ is a finite sequence of basic blocks $(v_0, v_1, ..., v_k)$, where $v_0 \in V_h$, $v_k \in V_h \cup V_e$ are the head and tail of σ and are denote as head(σ) and tail(σ), respectively. If $head(\sigma) == tail(\sigma)$, we say σ is an iterable path. The path condition of σ is the conjunction of the branch condition of each edge in the path and denote as θ_σ. We use $\mathcal{V}_\sigma$ to denote the value changes of the variables in the path σ and $\mathcal{V}_\sigma^n$ denote the variables after σ executes n times. $\theta(\sigma_i, \mathcal{V}_{\sigma_j}^n) \mapsto \{true, false\}$ represents if the path condition θ_{σ_i} is satisfiable or not after the path σ_j executes n times.



(a) Loop (b) CFG (c) PDA$_l$

Fig. 1. Unnested loop program

If θ_σ is satisfiable, σ is feasible, otherwise, it is infeasible. Obviously, the infeasible path will not be executed, so in this paper, we only consider feasible paths. In the following sections, we assume that the paths are all feasible.

The precondition of loop denoted as $pre(\mathcal{G})$ constrains the possible valuations for the variables in start node of CFG. The precondition of each path σ which constrains the possible valuations for the variables before executing the path σ.

Example 2. The loop in Fig. 1 has three paths in the CFG: $\sigma_1 = (b, c, e, b)$, $\sigma_2 = (b, c, f, b)$, $\sigma_3 = (b, d)$. σ_1 and σ_2 are the iterable paths. The path condition of σ_1 is $2x + y > 0 \wedge y > 0$. We use Θ_σ to denote the set of conditions of σ, e.g., $\Theta_{\sigma_1} = \{2x + y > 0, y > 0\}$. The loop in Fig. 2 has four paths: $\sigma_1 = (b, c)$, $\sigma_2 = (c, e, c)$, $\sigma_3 = (c, f, b)$, $\sigma_4 = (b, d)$. For each of the iterable path, we can convert it to a linear loop program. For example, in Fig. 1, the path σ_1, we can convert it to:

$$
\begin{aligned}
&while(2 * x + y > 0 \wedge y > 0)\{\\
&\quad x = -x - y;\\
&\quad y = -3 * y;\\
&\}
\end{aligned}
$$

Given a loop with CFG $\mathcal{G} = \{V, E, v_s, V_h, V_e, \iota\}$, the path dependency automaton (PDA$_l$) of this loop is $\mathcal{A} = \{S, T, init, accept\}$, where

- S is a set of states. Each state $\sigma \in S$ corresponds to a path in the loop.
- $T \subseteq S \times S$ is a set of transitions. $(\sigma_i, \sigma_j) \in T$, which represents $\exists n > 0$, s.t. $\theta(\sigma_j, \mathcal{V}^n_{\sigma_i}) == true \wedge tail(\sigma_i) == head(\sigma_j)$
- $init$ is a set of initial states in S and $accept$ is a set of accept states in S. An initial state is the firstly executed state and an accept state has no successors.

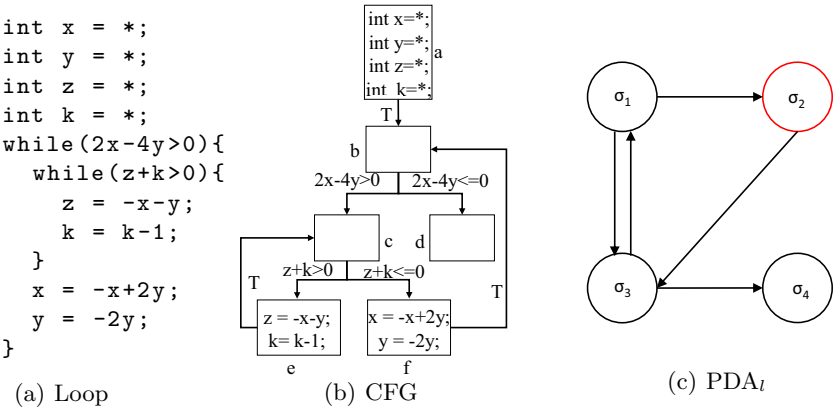


Fig. 2. Nested loop program

Example 3. Figure 1(c) shows the PDA$_l$ of the loop in Fig. 1(a). Where $S = \{\sigma_1, \sigma_2, \sigma_3\}$, $T = \{(\sigma_1, \sigma_2), (\sigma_1, \sigma_3), (\sigma_2, \sigma_3)\}$,
$init = \{\sigma_1, \sigma_2\}$, $accept = \{\sigma_3\}$.

2.3 The Structure of Loopster

Loopster [20] uses a divide-and-conquer approach to analyze the termination of a loop program. It is required three steps to determine whether the loop is terminated. 1) Extracting relevant statements from the loop program through the program slicing technique and constructing a control flow graph (CFG). 2) Using CFG to construct PDA and adopting monotonicity to analyze the termination for each path. 3) Analyzing the reachability of the non-terminating path in the loop and the termination of the cycle in PDA to determine the overall termination of the loop.

2.4 Termination of Linear Loop Program

Here we briefly introduce the algorithm proposed by Tiwari [18] to determine the termination of a linear loop program.

Given a homogeneous case of linear loop program

$$while(Bx > 0)\{x = Ax\} \tag{1}$$

We first transform it into

$$while(B^{'}y > 0)\{y = Jy\} \tag{2}$$

where J is a Jordan canonical form of A, P is the transition matrix from A to J, $A = PJP^{-1}$, $x = Py$, and $B^{'} = BP$. Then we only consider the Jordan blocks corresponding to the positive eigenvectors and get the program

$$\begin{aligned} &while(B_1^{'}y_1 + B_2^{'}y_2 + ... + B_r^{'}y_r > 0)\{ \\ &\quad y_1 = J_1 y_1; \\ &\quad y_2 = J_2 y_2; \\ &\quad ...; \\ &\quad y_r = J_r y_r; \\ &\} \end{aligned} \tag{3}$$

which termination is equivalent to the termination of the original loop program. Where the k-th condition after i-th iteration is

$$Cond(k,i) = B_{k,1}^{'}J_1^i + B_{k,2}^{'}J_2^i + ... + B_{k,r}^{'}J_r^i \tag{4}$$

Let $b_{m,n}$ be the n-th element of $B_{k,m}$, $Ind = \{11, 12, ..., 1n_1, 21, 22, ..., 2n_2, ..., r1, r2, ..., rn_r\}$. For all index $ind \in Ind$, if

$$\begin{cases} b_{ind}y = 0 \\ b_{indr}y > 0 \\ Cond(k,i)y > 0, i \in [0, \Pi_2(ind)] \end{cases} \tag{5}$$

are not satisfiable, the loop is terminating. Where $\Pi_1(ind)$, $\Pi_2(ind)$ are represent the left and right components of ind respectively (for example, $ind = 12$, $\Pi_1(ind) = 1$ and $\Pi_2(ind) = 2$), $b_{ind} = b_{\Pi_1(ind),\Pi_2(ind)}$ and $indr$ represent the right part of ind in Ind (for example if $ind = 12$, $indr \in \{13, 14..., 1n_1, 21, 22, ..., 2n_2, ..., r1, r2, ..., rn_r\}$), and y represent the initial value of each variables in the loop.

For the non-homogeneous case

$$while(Bx > b)\{x = Ax + c\} \tag{6}$$

we can convert it as the following form and then analyze the terminator of this homogeneous form.

$$while(\begin{pmatrix} B & b \end{pmatrix} x > 0) \left\{ x = \begin{pmatrix} A & c \\ 0 & 1 \end{pmatrix} x \right\} \tag{7}$$

3 Methodology

In this section, we introduce Loopster++ in detail. Loopster++ extends Loopster mainly in the following two parts (marked with red boxes in Fig. 3. 1) When constructing PDA_l, we use linear loop termination analysis to determine the termination of each path. We use the weakest post-condition to determine the dependencies between paths. 2) In the phase of loop overall termination analysis, we extend the linear loop termination algorithm to support the termination of the loop which has preconditions. For the cycle in PDA_l, we propose a new method to analyze the cycle termination.

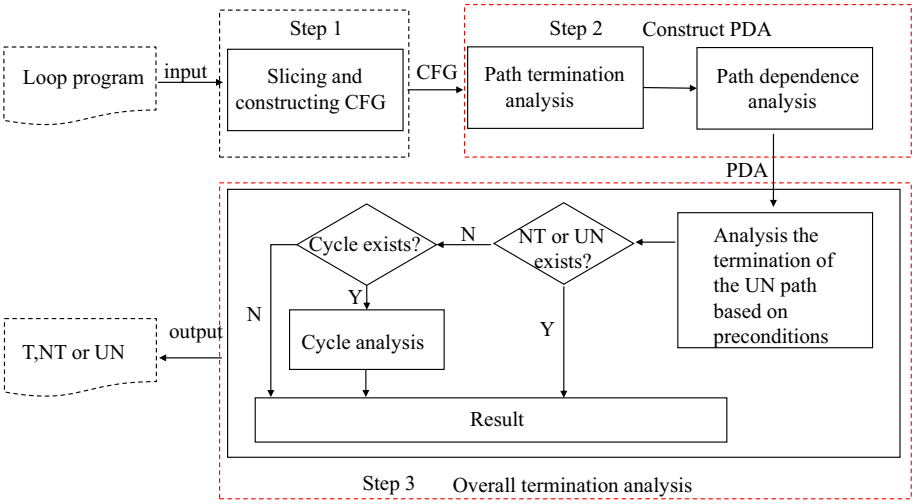


Fig. 3. Overview of Loopster++ (Color figure online)

3.1 Path Termination Analysis

The path can terminate if it can be executed within a limited number of times for any input, otherwise, the path is nonterminating. Obviously, a nonterminating path must be iterable, so we only consider the termination of iterable paths. For each iterable path, we first determine whether the path has a precondition. If there is no precondition, we only need to extract its path condition and value changes, and convert it into a simple linear loop program, and then use the algorithm introduced in Sect. 2.4 to determine the termination of this path. However, for the path with preconditions, we need slightly extend the algorithm introduced in Sect. 2.4. The specific methods are as follows:

From the formula (5) in the algorithm of Sect. 2.4, we can find that the algorithm to determine the termination is searching for whether there is an initial value y that satisfies the condition in formula (5). If it exists, the loop does not terminate. So we can set the range of the initial value of the corresponding variable here so that we can prevent the search for points other than the preconditions so as to determine the termination of the path containing the preconditions. For the precondition $pre(\sigma)$ of the path σ, we assume that all preconditions are in the form of $Cx \sim 0$, where C is the coefficient matrix of the variable x, $\sim \in \{<, >, \leq, \geq, ==\}$ is a relation symbol. We first limit the range of x by $Cx \sim 0$, and then determine the relationship between x and y according to the transformation from formula (1) to formula (2) in Sect. 2.4, so as to limit the search range in formula (5). If the path can be terminated under this restriction, we can determine that the path σ can be terminated under the condition $pre(\sigma)$.

Algorithm 1. PathTermAnalysis(σ,$pre(\sigma)$)

Input: σ: loop path
$pre(\sigma)$: the preconditions of σ
Output: $\{T, NT\}$
1: **if** $head(\sigma) \neq tail(\sigma)$ **then**
2: **return** T
3: **end if**
4: Construct *while* $Bx > 0$ *do* $x = Ax$
5: assume P is the transition matrix from A to J
6: $s = Solver()$
7: **if** $pre(\sigma)$ is not empty **then**
8: **for all** $precond \in pre(\sigma)$ **do**
9: $s.add(preocnd)$
10: **end for**
11: $s.add(y = P^{-1}x)$
12: **end if**
13: **return** isTerm(A, B, s)

Algorithm 1 shows how to analyze the termination property of a path. Obviously, it is always terminating if a path σ is not an iterable path ($head(\sigma) \neq tail(\sigma)$) (Lines 1–3). When a path σ is an iterable path, we first extract the path conditions and the variable changes and convert to the $while(Bx > 0)\{x = Ax\}$ form (Line 4). We initialize an empty SMT solver at line 6, and add the precondition in lines 7–11. At line 13, the function "isTerm" is an implementation of the algorithm in Sect. 2.4, this function will return whether the path is terminated.

Example 4. As shown in Fig. 1, for the path $\sigma_1 = (b, c, e, b)$, since $head(\sigma_1) == tail(\sigma_1)$, we construct it into a linear loop to check whether it can be terminated. We first extract its path condition $\theta_{\sigma_1} = 2x + y > 0 \wedge y > 0$ and extract its variable changes $\nu_{\sigma_1} = \{x = -x - y\,,\ y = -3y\}$. Then convert to the $while(Bx > 0)\{x = Ax\}$ form, where

$$\mathrm{A} = \begin{pmatrix} -1 & -1 \\ 0 & -3 \end{pmatrix} \qquad \mathrm{B} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

For this linear loop program, we can verify that it is terminated, so we return 'T' for the termination of σ_1.

3.2 Inter-Path Analysis

After analyzing the termination of each path, we need further to analyze whether a path can transit to another path. Obviously, for the path σ_i and σ_j if $tail(\sigma_i) \neq head(\sigma_j)$, then σ_i cannot be transferred to σ_j. Therefore, we only analyze the situation of $tail(\sigma_i) == head(\sigma_j)$. If the path σ_i only have one successor node, then σ_i can be transferred to σ_j. For other situations, we divide into two cases according to whether σ_i is iterable.

Algorithm 2. ComputeTran($\mathcal{G}$)

Input: $\mathcal{G} : CFG$
Output: T
1: $T = \{\}$
2: **for all** $(\sigma_i, \sigma_j) \in \{(\sigma_m, \sigma_n) | \sigma_m \in S \wedge \sigma_n \in S \wedge tail(\sigma_m) = head(\sigma_n) \wedge m \neq n\}$ **do**
3: **if** σ_i is termination $\wedge$ σ_i.outdegree == 1 **then**
4: $T = T \cup ((\sigma_i, \sigma_j))$
5: **else**
6: ${\theta_j}' = \text{substitue}(\theta_j, \sigma_i.\text{updates}())$
7: $\tau_{ij} = \theta_i \wedge {\theta_j}'$
8: **if** $head(\sigma_i) \neq tail(\sigma_i)$ **then**
9: **if** τ_{ij} is satisfiable **then**
10: $T = T \cup ((\sigma_i, \sigma_j))$
11: **end if**
12: **else**
13: ${\theta_i}' = \text{substitue}(\theta_i, \sigma_i.\text{updates}())$
14: $\tau_{ij} = \tau_{ij} \wedge \neg{\theta_i}'$
15: **if** τ_{ij} is satifiable $\vee$ $\theta(\sigma_j, \mathcal{V}^n_{\sigma_i})$ is satifiable **then**
16: $T = T \cup ((\sigma_i, \sigma_j))$
17: **end if**
18: **end if**
19: **end if**
20: **end for**
21: **return** T

– If the path σ_i is a one-time path, we only need to analyze whether there exists a set of variables that satisfy the path condition θ_i. After execute in σ_i, they satisfy the path condition θ_j. If it exists, we say that σ_i can be transferred to σ_j, otherwise, there is no dependency between the two paths.
– If the path σ_i is iterable, first of all, we analyze whether there exists a set of variables that satisfy the path condition θ_i. After execute in σ_i, they satisfy the path condition θ_j and they not satisfy the path condition θ_i. If it exists, we say that σ_i can be transferred to σ_j, otherwise, we use reachability analysis tools to determine whether these two paths can be transferred.

As shown in Algorithm 2, we present the method to analyze the dependency between every two paths. The necessary condition for σ_i to be transferred to σ_j is that the tail of σ_i is equal to the head of σ_j. Intuitively, if σ_i is termination and whose outdegree is 1, it can definitely be transferred to σ_j (Lines 3–4). At line 6, ${\theta_j}'$ represents the entry conditions after executing σ_i. At lines 8–11, we determine whether σ_i can be transferred to σ_j according to whether $\theta_i \wedge \theta_j$ can be satisfied. That is, after the program is executed, σ_i can satisfy the path condition of σ_j. If σ_i is an iterable path, we analyze whether σ_i can transfer to σ_j at lines 13–16. τ_{ij} is satisfiable means that there is a suitable input so that σ_i can be terminated and transferred to σ_j. $\theta(\sigma_j, \mathcal{V}^n_{\sigma_i})$ means σ_j can reach after σ_i execute n times.

Algorithm 3. merge($\mathcal{C}$)

Input: $\mathcal{C} = \{\sigma_1, \sigma_2, ..., \sigma_n\}$: the cycle constituted by $\sigma_1, \sigma_2, ..., \sigma_n$
Output: σ_{dummy}

```
1: valChange = {}
2: pathConditions = []
3: for all value ∈ C.values() do
4:     valChange[value] = value
5: end for
6: for all σ ∈ C do
7:     pathConditions + = substitue(σ.condition, valChange)
8:     for all update ∈ σ.updates() do
9:         valChange[update.left] = substitue(update.right, valChange)
10:    end for
11: end for
12: σ_dummy.setValChange(valChange)
13: σ_dummy.setConditions(pathConditions)
14: return σ_dummy
```

After analyzing the dependencies between paths, we need to analyze whether there are reachable non-terminating paths. If σ_n is non-terminating, we first collect all the precursors of σ_n, and then for each precursor, we use its weakest post-condition as the precondition of σ_n to determine the termination of σ_n. If σ_n terminates under each precursor, we say that σ_n can be terminated, otherwise, it cannot be terminated.

3.3 Cycle Analysis

If the PDA_l is acyclic and all path is termination, the loop is terminated. But when PDA_l has a cycle (e.g., the nested loop's PDA_l), the state in the cycle may be repeatedly executed alternately and result in non-termination. So we need further analysis of the termination of the cycle in PDA_l.

Definition 1. *Let* $\mathcal{C} = \{\sigma_1, \sigma_2, ..., \sigma_n\}$*, and it meets the following two conditions* 1) $\mathcal{C} \subset S$, 2) $\sigma_1, \sigma_2, ..., \sigma_n$ *constitute a strongly connected component (SCC) in* PDA_l*. We call* $\mathcal{C}$ *is a cycle in* PDA_l*.*

To analysis the cycle $\mathcal{C}$ in PDA_l, we divide the cycle into two categories as follows. Our analysis cycle termination method will be divided into type I cycle analysis and type II cycle analysis.

Definition 2. *If all paths in* $\mathcal{C}$ *are one-time paths, the cycle is type I cycle. Otherwise, the cycle is type II cycle.*

Theorem 1. *The termination of type I cycle is equivalent to the termination of the new path composed by each path of this cycle.*

In type I cycle $\mathcal{C}$ since all paths are one-time paths and all paths are executed in sequence, we can merge all paths in $\mathcal{C}$ one by one into a new path. And then, the termination of the cycle is equivalent to the termination of this new path.

Algorithm 3 merge the cycle $\mathcal{C}$ to a dummy path σ_{dummy}. In Theorem 1, we know that the termination of a type I cycle is determined by the path composed of each path in the cycle. Algorithm 3 introduces the specific method of merging each path in the cycle. First, we define an empty path σ_{dummy} and initialize the assignment of each variable to themselves (Lines 1–5). Then, we obtain the path condition of the visited path, and then replace all variables in the condition with the current variable assignment of σ_{dummy}, and add the replaced condition to the path condition of σ_{dummy} (Lines 6–7). Similarly, in lines 8–9, we also substitute the right part of the update statement on the visited path. Finally, we return to the merged path σ_{dummy} (Line 14). Example 5 introduces how to merge the type I cycle.

Example 5. The loop in Fig. 2 has the paths σ_1 and σ_3 which constitute a type I cycle. $\theta_{\sigma_1} = 2x - 4y > 0$ and $\theta_{\sigma_3} = z + k <= 0$, we conjunction of this two path conditions as a new loop condition. The value update in σ_1 is none and in σ_3 is $x = -x + 2y$, $y = -2y$. We construct a new loop as:

$$
\begin{aligned}
&while(2 * x - 4 * y > 0 \wedge z + k <= 0)\{\\
&\quad x = -x + 2 * y;\\
&\quad y = -2 * y;\\
&\}
\end{aligned}
$$

And then analyze the termination of this new loop, that is the termination of this cycle. If there is an update instruction on block c (example $\mathrm{z} = \mathrm{z} + 1$), the second condition of this new loop is $\mathrm{z} + \mathrm{k} + 1 <= 0$.

Definition 3. *All variables in $\mathcal{C}$ that can affect the exit path conditions are called key variables in $\mathcal{C}$.*

Theorem 2. *If a type II cycle can be reduced to a type I cycle, and the type I cycle is also a cycle in PDA_l, the termination of the type II type is equivalent to the termination of this type I cycle.*

For the type II cycle, our main idea is to determine whether the cycle can be reduced to a type I cycle, that is, whether all the iterable paths in this cycle can be removed. If it can be reduced, the termination of the type II cycle is the same as the termination of the type I cycle after the reduction. Here we find that there are two types of iterable paths that can be removed:

- For the iterable path σ, if all the variable updates in σ do not involve key variables, we say that σ can be removed.
- For the iterable path σ, if the variable that can affect the conditions of σ is only updated in σ and it is not updated outside σ, the path σ can be removed.

Algorithm 4. CycleAnalysis($\mathcal{C}$)

Input: $\mathcal{C} = \{\sigma_1, \sigma_2, ..., \sigma_n\}$ $\sigma_1, \sigma_2, ..., \sigma_n$ constitute a cycle
$pre(\sigma_1)$: precondition of σ_1
Output: $\{T, NT, UN\}$

```
1: if ∀σ_i ∈ C ∧ σ_i.iterable = false then
2:     σ = merge(C)
3:     return PathTermAnalysis(σ,pre(σ_1))
4: end if
   assume C_II is a set that all iterable path in C
5: for all σ ∈ C do
6:     if head(σ) == tail(σ) then
7:         C_II.append(σ)
8:     end if
9: end for
10: keyVars = C.getKeyVar();
        We use Θ_σ to denote the conditional of σ
11: for all σ ∈ C_II do
12:     if ∃θ_i ∈ Θ_σ only change in σ then
13:         C.del(σ)
14:         C_II.del(σ)
15:     else if ∀ var ∈ σ_i.vars() ∧ var ∉ keyVars then
16:         C.del(σ)
17:         C_II.del(σ)
18:     end if
19: end for

20: if C_II.empty()∧C.isTypeI()∧C ∈ PDA_l.cycles() then
21:     return CycleAnalysis(C)
22: end if
23: return UN
```

In type II cycle $\mathcal{C}$, we first collect all iterable paths in $\mathcal{C}$, and determine whether all iterable paths can be removed according to the above strategy. If the cycle $\mathcal{C}$ is reduced to a type I cycle after removing the iterable path, and we only need to analyze the termination of this cycle as a type I cycle. If there is still an iterable path in $\mathcal{C}$, we say that the termination of the cycle is unknown.

In Algorithm 4, we first check whether the input cycle $\mathcal{C}$ is a type I cycle. If it is a type I cycle, we merge its paths into a new path and then analyze the termination of this new path (Lines 1–3). After line 4 is the algorithm for analyzing the type II cycle. First, we collect all the iterable paths (Lines 5–7) in the cycle $\mathcal{C}$ to $\mathcal{C}_{II}$ and then analyze whether these iterable paths can be removed and remove those removable paths (Lines 10–19). After removing the removable paths, we need to check whether the set of iterable paths $\mathcal{C}_{II}$ is empty; that is, the cycle can be reduced to a type I cycle. If it is empty and $\mathcal{C}$ be reduced to a type I cycle, we return the termination of the new cycle (Lines 20–21); otherwise, we can not analyze the termination of the cycle $\mathcal{C}$ and return "UN" (Line 23).

Table 1. Experimental result

	Loopster++				Ultimate automizer				CPAchecker				2LS			
	CTT	CFF	CUN	COT	CTT	CFF	CUN	COT	CTT	CFF	CUN	COT	CTT	CFF	CUN	COT
Nested	19	2	9	0	26	3	0	1	23	3	3	1	20	3	7	0
Unnested	64	15	11	0	70	15	5	0	57	12	17	4	60	8	22	0
Total	83	17	20	0	96	18	5	1	80	15	20	5	80	11	29	0
Time(s)	344.03				1838.40				9229.49				1864.05			

Example 6. The loop in Fig. 2 has the type II cycle $\mathcal{C}_2 = \{\sigma_1, \sigma_2, \sigma_3\}$. σ_2 is the iterable path where the path condition of σ_2 is $z + k > 0$ and z, k only change in σ_2, so we remove σ_2 in $\mathcal{C}_2$. In fact, in this loop, the key variables are x, y. In this iterable path, the value of the key variable will not be changed, so the path can also be removed by the method in lines 15–17 of the Algorithm 4. After remove, $\mathcal{C}_2 = \{\sigma_1, \sigma_2\}$ is a type I cycle. Finally, we use the method of analysis the type I cycle to analysis the termination of $\mathcal{C}_2$.

4 Implementation and Evaluation

We implement the Loopster++ based on LLVM 8.0 [13], Seahorn [10] and SMT solver Z3 [15]. We also implement Tiwari's linear loops program termination decision algorithm to determine the termination of the path in our tool. To evaluate the effectiveness and performance of loopster++, we compare our tool with the state-of-the-art tools.

4.1 Effectiveness of Loopster++

To evaluate the effectiveness and performance of loopster++, we compare our tool against the other three termination analysis tools, namely Ultimate Automizer [11], CPAchecker [4], and 2LS [6], which are the top three ranking in the termination analysis section of 9th Competition on Software Verification (SV-COMP 2020) [1]. We selected all linear loops with terminating properties in termination-crafted, termination-crafted-lit, and termination–restricted-15 benchmarks from the category of SV-COMP 2020. There are 120 of these loop programs in total, 30 of them are nested loops and 90 are non-nested loops. The termination of these 120 loop programs is known that 99 of them are termination and 21 are non-termination. All experiments run on a Ubuntu 18.04.4 LTS system in a virtual machine with Intel Core i7-4790 CPU @ 3.60 GHz (1 core) and 4.7 GB memory.

Table 1 shows the experimental results, and columns CTT, CFF and CUN represent the number of termination loops, non-termination loops and unknown results separately. COT summarizes the number of programs that analyzed time out and Time represents the total time spent analyzing 120 programs. In this table, we measure the time in second and we set the timeout as 600 s.

In Table 1, the result shows that our tool can correctly handle 100 programs with the shortest time (344.03 s). Ultimate Automizer can correctly analyze 114 programs in 1838.40 s and have 1 program out of time. CPAchecker has correctly analyzed 95 programs, and 5 programs have time out, therefore CPAchecker takes the longest time (9229.49 s). 2LS takes 1864.05 s to analyze all programs and has correctly analyzed 91 programs. However, when analyzing the termination of linear loops, 20 of them cannot get correct results, which have exceeded CPAchecker and 2LS. In summary, Loopster++ is second only to UA in the number of programs handled correctly. But Loopster++ is better than the other three tools in terms of solution time. At least 5 times faster than the most efficient Ultimate Automizer among them.

In these 120 loop programs, we recorded the methods Loopstr++ used when analyzing their termination. Among them, 63 programs can directly determine the overall termination of the loop by analyzing the path termination and the dependence between the paths. There are 26 programs that need to use preconditions to determine the termination of the path in the process of analyzing the termination. There are 37 programs that will generate cycles during the analysis process, of which 20 programs generate loops and we give unknown results.

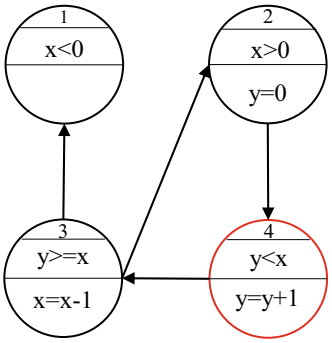
Table 2. The unknown results given by Loopster++

Loop programs	Is nested
a.01.c	Yes
AliasDarteFeautrierGonnord-SAS2010-loops.c	Yes
AliasDarteFeautrierGonnord-SAS2010-wcet2.c	Yes
java_Nested.c	Yes
McCarthy91_Iteration.c	Yes
NO_03.c	Yes
PastaA1.c	Yes
Urban-WST2013-Fig2-modified1000.c	Yes
Urban-WST2013-Fig2.c	Yes
AliasDarteFeautrierGonnord-SAS2010-cousot9.c	No
AliasDarteFeautrierGonnord-SAS2010-speedpldi2.c	No
AliasDarteFeautrierGonnord-SAS2010-speedpldi3.c	No
c.03.c	No
Flip2.c	No
Gothenburg-1.c	No
Gothenburg_v2-1.c	No
McCarthyIterative.c	No
NO_13.c	No
PastaC3.c	No
UpAndDownIneq.c	No

As shown in Table 2, for the unknown result given by our tool, which has 9 nested loops and 11 unnested loops. The reason why these loops cannot be given

```
int main() {
  int c = 0;
  int x = *;
  int y = *;
  while (x > 0) {
    y = 0;
    while (y < x) {
      y = y + 1;
    }
    x = x - 1;
  }
  return 0;
}
```

(a) Loop



(b) PDA$_l$

Fig. 4. Loop program a.01.c

the correct results is that there is a cycle in PDA$_l$, and we cannot analyze the termination of the cycle by Algorithm 4 in Sect. 3. We summarize the reasons as follows.

- For unnested loop programs, the cycle in PDA$_l$ is composed of $if...else...$; for example $while(i > 0)\{if(j > 0)\{j = j - 1;\}else\{j = N; i = i - 1;\}\}$, in this loop, the paths formed by the branches of if and $else$ are all iterable paths, and the interdependence of these two paths forms a type II cycle; and we cannot reduce it to a type I cycle.
- For nested loops, there are still complex dependencies between inner loops and outer loops that constitute the cycle, and we cannot be sure of termination. For example, In Fig. 4, the loop program a.01.c is terminating, but our tool cannot obtain the correct result. Figure 4(b) is the PDA$_l$ of this loop program. From this PDA$_l$, we can see that σ_4 is an iterable path. When we remove σ_4, we cannot get the type I cycle in the PDA$_l$. So we cannot determine whether the cycle is terminated, and the result of the unknown is given. Therefore, if we can find better ways to analyze loops (for example, find more types of cycles that can be analyzed or using the ranking function technique to handle cycle), our tools can be more complete.

4.2 Performance of Loopster++

To evaluate the performance of Loopster++, we compare the performance of our tool with the other three tools in terms of the time it takes to verify the termination of each program. At the same time, we analyzed why our tool has a good performance.

Figure 5 shows the detailed analysis time of each tool. The blue dot is the time distribution of our tool Loopster++. We can see that Loopster++ can verify the termination of the benchmark within 5 s. Note that, 2ls cannot work efficiently

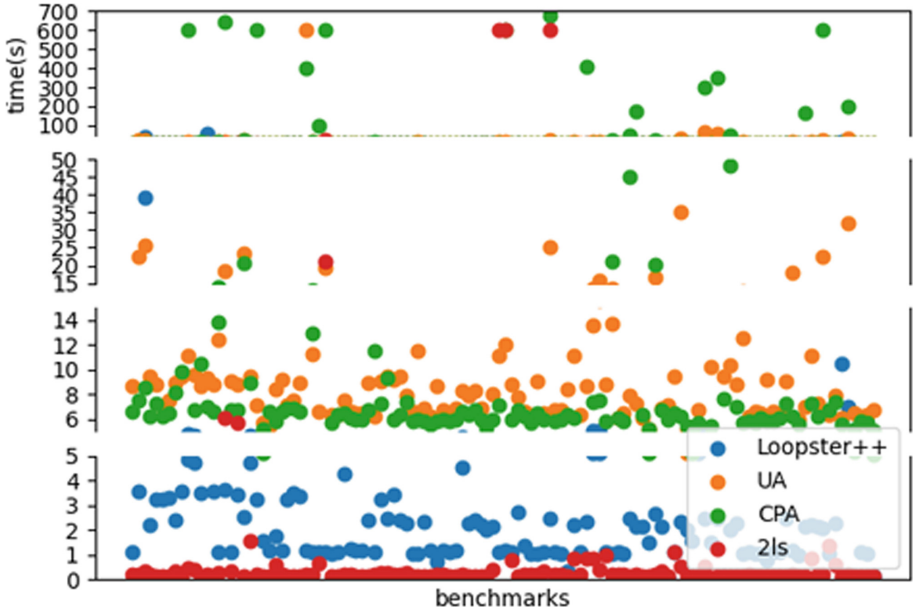


Fig. 5. Detail run time

on all benchamrks, and some benchmarks takes a very long time (which is the main reason for the high total time), even if it can verify most benchmarks in a short time. As shown in Fig. 5, we can also find that our tool is faster than CPAchecker and Ultimate Automizer for the verification of each program. In the following, we analyze and discuss the possible reasons why our tool can steadily prove the termination.

- We separate the analysis of each path and the analysis of dependencies between paths, which simplifies the problem to a certain extent. This determines that our tool analysis of the termination of the program will not take much time.
- When our tool analyzes program termination, the main part of the time is spent on the path termination analysis. As for these benchmarks, the number of path branches is about the same, so it takes us about the same time to analyze the termination of each program.

From Fig. 5, we can find that it takes a long time for Loopster++ to analyze its termination for some loop programs. Our experimental results shows that these loop programs with a sudden increase in time often have the characteristics of more conditions or more variables. For loops with many conditions, the *Ind* Sect. 2.4 will produce more groups of possible non-terminating points when analyzing the termination of the path. For loops with many variables, a higher-dimensional variable update matrix can be generated by Loopster++, which will

increase the probability of a greater number of positive eigenvalues and expand the Ind set.

5 Relate Work

In this section, we discuss the related work on the termination proving on the linear loop programs.

Dams et al. first proposed a method to automatically synthesize the ranking functions for linear loops [9]. Michael and Henny [7,8] presented an algorithm to generate the linear ranking functions by manipulating polyhedral cones. Podelski and Rybalchenko [17] presented a complete method for the linear ranking functions through the relationship between linear inequalities. Ben-Amram and Genaim [3] studied the complexity of generating the linear ranking functions for a linear loop and they proved that the complexity is coNP-complete when the variable range over the integers. Leike and Heizmann [14] introduced the notion of linear ranking templates, and based on this to study the constraint-based synthesis of termination arguments for linear loop programs.

However, for the termination of loop programs, the ranking function does not necessarily exist (this is just a sufficiently unnecessary condition). There are other people who pay more attention to study a decidable class of loops. Tiwari [18] used algebraic theory to propose a method for finding nonterminating points (if it does not exist, the program will terminating) and he also proved that it is decidable over the reals $\mathbb{R}$. Braverman generalized the technique presented in [18] and they presented a decision procedure for simple homogeneous linear loop programs over the integers [5]. If the update matrix A of a simple linear loop program can be diagonalized, Ouaknine et al. [16] proposed how to decide its termination.

Compared with the above techniques [3,7–9,14,17], our approach does not need to synthesize the ranking functions. We pay attention to a class of decidable loops to ensure that the termination of each path can be analyzed. Unlike the techniques in [5,16,18], our approach focuses on the termination of multi-path linear loops.

As an extension of Loopster [20], our work is also related to Loopster. Loopster used the monotonicity theory and the dependence between paths to determine the termination of a loop program. The difference is that our approach uses the theory of linear loop programs to determine the termination of each path. At the same time, the way of handling dependencies between paths and the method of analyzing the cycle is also different.

6 Conclusion

In this paper, we extend Loopster as Loopster++ to analyze the termination of multi-path linear loops. We determine the termination of the loop by analyzing the termination of the path, the reachability of each two paths, and the termination of the cycle in PDA_l. Finally, we implemented our approach and designed

experiments to compare our tools with the state-of-the-art tools. The empirical results show that our approach is effective to achieve high accuracy and is more efficient with a shorter analysis time. In the future, we plan to extend our approach more complete and make it to support more types of loops, such as nonlinear loop programs.

Acknowledgment. This work has partially been sponsored by the National Science Foundation of China (No. 61872262).

References

1. 9th competition on software verification (2020). https://sv-comp.sosy-lab.org/2020/
2. CVE-2009-1890 (2020). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1890
3. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. SIGPLAN Not. **48**(1), 51–62 (2013). https://doi.org/10.1145/2480359.2429078
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_51
5. Braverman, M.: Termination of integer linear programs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 372–385. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_34
6. Chen, H., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 53–64 (2015)
7. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_36
8. Colóon, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_6
9. Dams, D., Gerth, R., Grumberg, O.: A heuristic for the automatic generation of ranking functions. In: Workshop on Advances in Verification, pp. 1–8 (2000)
10. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
11. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53
12. Larraz, D., Oliveras, A., Rodriguez-Carbonell, E., Rubio, A.: Proving termination of imperative programs using Max-SMT. In: 2013 Formal Methods in Computer-Aided Design, FMCAD 2013, pp. 218–225 (2013)
13. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis transformation. In: International Symposium on Code Generation and Optimization, CGO 2004, pp. 75–86 (2004)

14. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 172–186. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_12
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Ouaknine, J., Pinto, J.A.S., Worrell, J.: On termination of integer linear loops. In: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, pp. 957–969. Society for Industrial and Applied Mathematics, USA (2015)
17. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_20
18. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_6
19. Xie, X., Chen, B., Liu, Y., Le, W., Li, X.: Proteus: computing disjunctive loop summary via path dependency analysis. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 61–72. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2950290.2950340
20. Xie, X., Chen, B., Zou, L., Lin, S.W., Liu, Y., Li, X.: Loopster: static loop termination analysis. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 84–94. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3106237.3106260