This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

Automatic Loop Summarization via Path Dependency Analysis

Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, Xiaohong Li

Abstract—Analyzing loops is very important for various software engineering tasks such as bug detection, test case generation and program optimization. However, loops are very challenging structures for program analysis, especially when (nested) loops contain multiple paths that have complex interleaving relationships. In this paper, we propose the path dependency automaton (PDA) to capture the dependencies among the multiple paths in a loop. Based on the PDA, we first propose a loop classification to understand the complexity of loop summarization. Then, we propose a loop analysis framework, named Proteus, which takes a loop program and a set of variables of interest as inputs and summarizes path-sensitive loop effects (i.e., disjunctive loop summary) on the variables of interest. An algorithm is proposed to traverse the PDA to summarize the effect for all possible executions in the loop. We have evaluated Proteus using loops from five open-source projects and two well-known benchmarks and applying the disjunctive loop summary to three applications: loop bound analysis, program verification and test case generation. The evaluation results have demonstrated that Proteus can compute a more precise bound than the existing loop bound analysis techniques; Proteus can significantly outperform the state-of-the-art tools for loop program verification; and Proteus can help generate test cases for deep loops within one second, while symbolic execution tools KLEE and Pex either need much more time or fail.

Index Terms—Disjunctive Loop Summary, Path Dependency Automaton, Path Interleaving

1 Introduction

Analyzing loops is very important for various software engineering tasks, e.g., bug detection, test case generation, and program optimization. However, loop analysis is one of the most challenging tasks in program analysis, especially when (nested) loops contain multiple paths that have complex interleaving relationships. Specifically, loops are the "Achilles' heel" of program verification [1], and a key bottleneck for scaling symbolic execution [2], [3].

1.1 Existing Work

Three kinds of techniques are mainly used for analyzing loops, namely *loop unwinding*, *loop invariant inference* and *loop summarization*. Loop unwinding unrolls the loop with a fixed number of iterations. This technique is simple but cannot reason about the program behaviors beyond the unwinding bound. A loop invariant is a property that holds in each loop iteration. The limitations are that, applications typically rely only on the sufficiently strong loop invariants [1] that might be difficult to infer especially for complex loops; and commonly-used fixpoint-based invariant inference [4] is iterative and sometimes time-consuming.

- *X. Xie and X. Li are with the School of Computer Science and Technology, Tianjin University, China, 300350. E-mail: {xiexiaofei, xiaohongli}@tju.edu.cn*
- *B. Chen is with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China, 201203. E-mail: bhchen@fudan.edu.cn*
- *L. Zou and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, 639798. E-mail: {lzou, yangliu}@ntu.edu.sg*
- *W. Le is with the Department of Computer Science, Iowa State University, USA, 50011. E-mail: weile@iastate.edu*

Compared to loop invariants, loop summarization provides a more accurate and more complete comprehension for loops [1], [2], [5], [6], [7] in terms of loop summary. A loop summary captures the relationship between the inputs and outputs of a loop as a set of symbolic constraints. Therefore, we can replace a loop fragment with its loop summary during program analysis. For example, we can use a loop summary to verify program properties after a loop; and we can use it to direct test case generation in symbolic execution.

1.2 Challenges

Several loop summarization techniques [2], [5], [6], [7] have been proposed, however, challenges still remain for *multi-path loops* (loops that contain branches) and the *interleaving execution* among multiple paths. For example, the loop summarization techniques in [2], [5] handle *single-path loops* where variables are modified by a constant in each loop iteration. The recent advances of loop analysis [6], [7] can summarize some multi-path loops but they do not consider the interleaving pattern of the paths. The multi-path loops can be nested or unnested loops, and their executions can be sequential or interleaving. Specially, *nested loops* are challenging because 1) nested loops are all multi-path loops because there are multiple paths in outer loops and inner loops; and 2) nested loops are usually interleaving execution among the paths due to the nesting between the inner loops and outer loops. Hence, the existing approaches either cannot handle multi-path loops or do not consider the interleaving pattern. The goal of this paper is to reason about the interleaving of multiple paths in a nested or unnested loop and generate a *disjunctive loop summary* (DLS) for such multi-path loops.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

As an example, the `while` loop in Fig. 1(a) contains an `if-else` branch, which makes it a multi-path loop; and the computation in the `if` and `else` branches impacts the evaluation of the `if` condition, leading to the interleaving of the two paths in the loop. The initial values of the variables x, z and n determine the possibilities of interleaving between the `if` and `else` branches, which produce different effects after the loop execution. Let x, z and x', z' be the values before and after the loop execution respectively. If the initial values satisfy $x \geq n$, the loop effect is $x' = x \wedge z' = z$; if $x < n \leq z$, the loop effect is $x' = n \wedge z' = z$; and if the loop starts with $x < n \wedge z < n$, the loop effect is $x' = z' = n$. Therefore, loop summarization should consider all interleaving possibilities and a precise summary should be a *disjunction* that includes all possible loop executions due to the different initial values of the variables. Hence, the DLS for Fig. 1(a) is $(x \geq n \wedge x' = x \wedge z' = z) \vee (x < n \leq z \wedge x' = n \wedge z' = z) \vee (x < n \wedge z < n \wedge x' = z' = n)$. Comparing with the loop summary from [2], [5], [6], [7], DLS considers each possible pattern of interleaving and is more specific and fine-grained[1].

1.3 The Proposed Approach

This paper accomplishes three tasks to advance the state of the art in loop analysis. First, we propose a *path dependency automaton* (PDA) to model the relationships between multiple paths of unnested and nested loops. Second, we propose a classification to understand the difficulty of loop summarization. The loop classification defines what types of multi-path loops we can handle precisely, what types of multi-path loops we can handle with approximation, and what types of multi-path loops we cannot yet handle. Last, we develop a loop analysis framework, named Proteus[2], to summarize the loop effect for each loop type based on the PDA. Proteus takes a loop code fragment and a set of variables of interest as inputs to compute the DLS. The DLS is a disjunction of a set of path-sensitive loop effects on the variables of interest.

Proteus generates a fine-grained loop summary in three steps. The first step applies program slicing on the loop according to the variables of interest such that some irrelevant paths can be reduced in the loop. The second step is a novel technique where a PDA is constructed from the control flow graph (CFG) of the loop to model the path interleaving. Each state in the PDA corresponds to a path in the CFG; and transitions in the PDA capture the dependencies of the paths. The last step traverses the PDA to summarize the effect of each feasible trace in the PDA (each trace represents a possible execution in the loop). The final result, i.e., DLS, is a disjunction of the summaries for all possible executions of the loop.

We have implemented Proteus and evaluated the usefulness of DLS by applying it to loop bound analysis, program verification and test case generation. We computed the loop bound for 8,782 loops from five open-source projects. The results indicate that Proteus can compute a more precise loop bound than the existing techniques [8], [9], [10], [11]. We also applied DLS to program verification on 169 programs from two well-known benchmarks. The results show that Proteus can summarize 136 (80.47%) programs; and for these summarized programs, Proteus can help verify 133 (97.80%) programs correctly, while SMACK+Corral [12], which achieved the highest correct rate in SV-COMP'16 [13], can only verify 106 (77.94%) programs correctly. Besides, Proteus only took 64 seconds, while SMACK+Corral took more than 7 hours. We also applied DLS to test case generation, and compared symbolic execution tools KLEE [14] and Pex [15] with Proteus. Our results show that Proteus can generate test cases for the deep loops (large number of iterations) in one second while KLEE either times out or needs much more time and Pex often throws an exception.

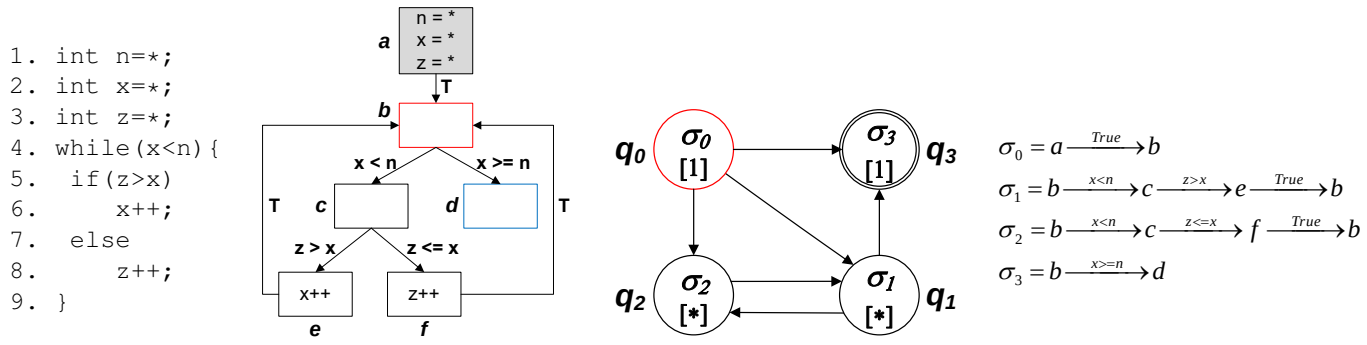
This work is a substantial extended version of our previous work [16]. In addition to a more detailed and systematic description of the approach, we extend the previous work to be more general in the following aspects.

1) we extend our framework to support nested loops which cannot be modeled (Section 2), classified and summarized in the previous work. Based on our extended PDA in [17], we improve the classification (Section 3) and summarization (Section 5) on both unnested and nested loops.
2) we extend the summarization algorithm to support more complex variables (Section 4.2). In the previous work, we can only summarize the variables which are changed with a constant (i.e., the update values of each variable is Arithmetic Sequence). In this work, we extend the definition of induction variable, and can summarize the variables (Algorithm 1, 2 and 3) whose n-th term can be calculated with the domain knowledge of mathematics. For example, the Constant Sequence, the Geometric Sequence and the combined Sequences can also be handled. Hence, the new Type 1 loops we can summarize in this work contain part of the Type 2 and 3 loops in the previous work.
3) we extend the summarization algorithm to handle more periodic cycles (Algorithm 3). In the previous work, the execution count of each state in each execution of the periodic cycle must be the same. In this work, we extend that the cycle is periodic if the execution count of each state has a regular change, which can be determined (e.g., Arithmetic Sequence, Geometric Sequence and so on).
4) we extend the summarization algorithm to handle some connected cycles by analyzing the dependencies among the cycles (Section 5.3), while the previous work does not handle and summarize the PDA that contains connected cycles.
5) we provide more detailed proofs for the soundness of the approach (Theorem 1, 2 and 3).
6) we add more benchmarks and rerun all the experiments (Section 7). For the bound analysis application, we classify and compute bounds for the nested loops in the five open-source projects. For program verification, we add one new benchmark that contains nested loops and more complex unnested loops. For test case generation, we choose several programs from the new benchmark as the target for generating test cases.

1. In this paper, we say DLS is more fine-grained since DLS summarizes for each possible trace of the loop and it is a disjunction of all trace summaries.

2. A Greek god who can foretell the future.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering



(a) Loop [9] (b) CFG (c) PDA

Fig. 1: Unnested Loop

1.4 Contributions

In summary, the main contributions of this work are:

1) we propose a path dependency automaton to capture the dependency and interleaving of the paths in multi-path loops;
2) we propose a classification for multi-path (nested) loops of four types to understand the complexity of loop summarization;
3) we propose a loop analysis framework, Proteus, to compute disjunctive loop summary based on the path dependency automaton; and
4) we conduct an evaluation to demonstrate the usefulness of disjunctive loop summaries in three important applications.

The rest of this paper is structured as follows. Section 2 defines the path dependency automaton. Section 3 presents the loop classification and the overview of our loop analysis framework. Section 4 introduces the construction of path dependency automaton. Section 5 and 6 elaborate our summarization approach for different types of loops. Section 7 evaluates the usefulness of our approach. Section 8 reviews the related work before Section 9 draws the conclusions.

2 Loop Modeling

In this section, we first introduce the concepts of control flow graph and loop paths, and then we propose the path dependency automaton.

2.1 Preliminaries

Let D be a finite integer domain and $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables ranging over D. An *atomic condition* over X is in the form of $g(x_1, x_2, \ldots, x_n) \sim b$, where $g : D^n \mapsto D$ is a function that performs the integer operations on X, $\sim \in \{=, <, \leq, >, \geq\}$ and $b \in D$. For the operator $\neq$, we transform the condition to $e > 0 \vee e < 0$. We use P_X to denote the set of all possible atomic conditions over X. A condition over X can be a Boolean combination of atomic conditions over X.

A loop can be modeled by a control flow graph (CFG), as formulated in Definition 1. In this paper, we assume the loops always terminate. The summarization of non-terminating loops will be discussed in Section 5.5.

Definition 1. A *control flow graph of a loop* is a tuple $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$, where (L, E) is a finite directed graph; L is a set of basic blocks, each of which contains a sequence of assignment instructions; $E \subseteq L \times P_X \times L$ is a set of directed edges connecting the basic blocks; $l_{pre} \in L$ is the pre-header after which the loop enters into the entry block; $L_h \subset L$ is a set of header blocks; and $L_e \subset L$ is a set of exit blocks.

- Each edge $(l, p, l') \in E$ represents that the basic block l' can be executed after l if the condition p is satisfiable. For simplicity, we use $l \xrightarrow{p} l'$ to denote (l, p, l').
- $l \in L$ dominates $l' \in L$ if every path from l_{pre} to l' goes through l.
- $l \in L_h$ is a header block if there is an edge $(l', p, l) \in E$ and l dominates l'.
- $l \in L_e$ is an exit block if there is an edge $(l', p, l) \in E$ where l' is in the loop and l is not in the loop.

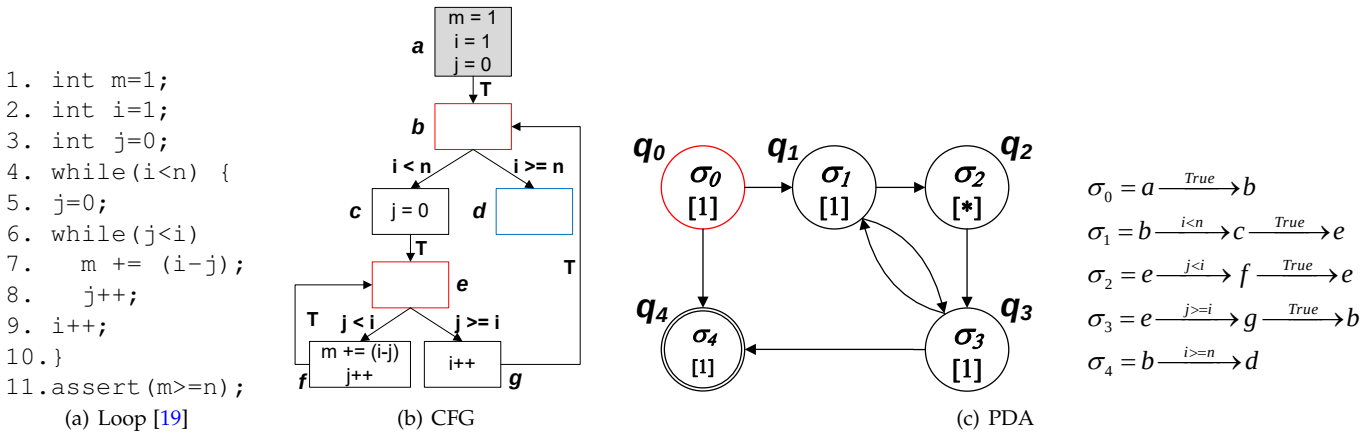
Let wp be the weakest precondition operator based on Hoare Logic [18]. wp takes the instruction s and a postcondition Q as inputs, and returns the weakest precondition of s with respect to Q, denoted as $wp(s, Q)$. In this paper, we use the assignment and sequence rules to compute the weakest precondition:

$$wp(x := E, Q) = Q[x \leftarrow E]$$
$$wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$$

Definition 2. Given a CFG $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$ of a loop, a path (denoted as σ) is a sequence of edges, $l_0 \xrightarrow{p_0} l_1 \xrightarrow{p_1} \ldots \xrightarrow{p_{k-1}} l_k$, where $l_0 \in L_h \cup \{l_{pre}\}$ is the head of σ, denoted by $head(\sigma)$; $l_k \in L_h \cup L_e$ is the tail of σ, denoted by $tail(\sigma)$; and $\forall 1 \leq i < k, l_i \notin L_h \cup L_e$. The weakest triggering condition of σ (denoted as θ_σ), i.e., the weakest condition under which σ can be executed, is computed as $wp(l_0, p_0) \wedge wp(l_0; l_1, p_1) \wedge \ldots \wedge wp(l_0; \ldots; l_{k-1}, p_{k-1})$, where each l_i in wp represents a sequence of instructions in the basic block l_i. We use $\prod_{\mathcal{G}}$ to denote the set of all paths in $\mathcal{G}$.

In this paper, we will use *path condition* to represent the weakest triggering condition of a path. A path σ is called an *iterative path* if $head(\sigma) = tail(\sigma)$; otherwise it is called a *one-time path*. Intuitively, the loop execution consists of the interleaving of multiple iterations of paths. During the loop execution, an iterative path can be consecutively executed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering



(a) Loop [19] (b) CFG (c) PDA

Fig. 2: Nested Loop

for multiple times. A one-time path can only be executed once and then one of the other paths is executed. We use σ^k to denote the k consecutive executions of path σ, i.e, the path σ is executed for k times consecutively. We define the function $f_\sigma(X,k)$ that computes the value of the variables X after k consecutive executions of path σ. For example, if x is updated as $x := x+1$ in each execution of σ, we compute $f_\sigma(x,10) = x+10$. For the sake of simplification, we also use X_{σ^k} to represent $f_\sigma(X,k)$.

Example 1. Fig. 1 and 2 show the examples of an unnested and a nested loop, respectively. Consider Fig. 2(b) which shows the control flow graph of the nested loop in Fig. 2(a). In this CFG, there are seven basic blocks named from a to g, and eight edges. a is the pre-header from which the loop program executes the loop, b and e are the header blocks, and d is the exit block. Specially, the basic blocks b, e and d contain no instructions. The edge $b \xrightarrow{i<n} c$ represents that c can be executed after b if the condition $i < n$ holds. T represents `True`, which means the edge is always feasible. In the CFG, there are five paths that are shown in Fig. 2(c). Among them, σ_2 is an iterative path, and the others are one-time paths. Intuitively, σ_2 can be executed many times before the execution of σ_3. σ_1 is a one-time path and it can only be executed once before the execution of σ_2 or σ_3. Similarly, in Fig. 1(b), there are four paths. In each execution of path σ_1, x increases by one. Thus we can compute $f_{\sigma_1}(x,k) = x+k$ (i.e., $x_{\sigma_1{}^k} = x+k$).

2.2 Path Dependency Automaton

Loop execution is a sequence of execution of the paths, which represents the interleaving among the paths. To model the interleaving among different paths in the loop, we propose the path dependency automaton (PDA) as follows.

Definition 3. Given a loop with its CFG $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$, the *path dependency automaton* (PDA) is a tuple $\mathcal{A} = (Q, \mathcal{L}, q_0, accept, T)$, where

- $Q = \{q_0, \ldots, q_n\}$ is a finite set of states.
- $\mathcal{L} : Q \to \prod_{\mathcal{G}}$ is a function that maps each state $q \in Q$ to a path $\mathcal{L}(q) \in \prod_{\mathcal{G}}$. We also use the notation $\mathcal{L}_q$ to represent $\mathcal{L}(q)$.
- q_0 is the initial state, where $head(\mathcal{L}_{q_0}) = l_{pre}$.
- $accept = \{q \in Q \mid tail(\mathcal{L}_q) \in L_e\}$ is a set of accepting states.
- $T = \{(q,q') \in Q \times Q \mid tail(\mathcal{L}_q) = head(\mathcal{L}_{q'}) \wedge \mathcal{L}_q \neq \mathcal{L}_{q'} \wedge (\exists k\colon (\bigwedge_{0\le i<k} \theta_{\mathcal{L}_q}[X_{\mathcal{L}_q{}^i}/X]) \wedge \theta_{\mathcal{L}_{q'}}[X_{\mathcal{L}_q{}^k}/X]$ is satisfiable$)\}$ is a finite set of transitions.

A state in the PDA corresponds to a path in the CFG of the loop. The initial state represents the path that is first executed. An accepting state represents the path whose tail is an exit block, and the loop terminates after the execution of that path. The semantics of one transition $(q,q') \in T$ is that the path $\mathcal{L}_{q'}$ can be executed after finite ($k \ge 1$) executions of the path $\mathcal{L}_q$. Note that, one state q can only transit to another state q' when the head node of $\mathcal{L}_{q'}$ is equal to the tail node of $\mathcal{L}_q$.

$\theta_{\mathcal{L}_q}[X_{\mathcal{L}_q{}^k}/X]$ represents that the variables in the path condition of $\mathcal{L}_q$ are substituted with the value after k executions of the path $\mathcal{L}_q$. For example, after k executions of σ_1 in Fig. 1(b), we can compute $x_{\sigma^k} = x+k$, the path condition $x < n \wedge z > x$ becomes $x+k < n \wedge z > x+k$. The transition (q,q') is feasible if *the path condition $\theta_{\mathcal{L}_q}$ always holds after each of the first $k-1$ executions of $\mathcal{L}_q$, and $\theta_{\mathcal{L}_{q'}}$ will hold after k executions of path $\mathcal{L}_q$.* The existence of k ensures $\mathcal{L}_{q'}$ can be executed eventually. Note that, if $\mathcal{L}_q$ is a one-time path, then the path $\mathcal{L}_q$ can only be executed once (i.e., $k=1$) before transiting to $\mathcal{L}_{q'}$. Specially, the accepting state has no successor (i.e., there is no transition starting from the accepting state) since the state ends with an exit block.

Definition 4. Given a loop with its PDA $\mathcal{A} = (Q, \mathcal{L}, q_0, accept, T)$, $E_{\mathcal{A}} = \{(q_0, \ldots, q_n) \mid q_n \in accept \wedge \exists k_0, \ldots, k_{n-1}\colon \bigwedge_{0\le i<n}((\bigwedge_{0\le j<k_i} \theta_{\mathcal{L}_{q_i}}[X_{\mathcal{L}_{q_i}{}^j}/X]) \wedge \theta_{\mathcal{L}_{q_{i+1}}}[X_{\mathcal{L}_{q_i}{}^{k_i}}/X])$ is satisfiable, where $X_{\mathcal{L}_{q_0}{}^0} = X \wedge \forall 0 < i \le n : X_{\mathcal{L}_{q_i}{}^0} = X_{\mathcal{L}_{q_{i-1}}{}^{k_{i-1}}}\}$ is a set of feasible traces.

We assume the loop always terminates, and thus each trace is finite and ends with an accepting state. The trace $(q_0, \ldots, q_n)$ is feasible if the transitions $(q_0,q_1), \ldots, (q_{n-1},q_n)$ are feasible. The conditions in Definition 4 include two parts: 1) $\bigwedge_{0\le j<k_i} \theta_{\mathcal{L}_{q_i}}[X_{\mathcal{L}_{q_i}{}^j}/X]) \wedge \theta_{\mathcal{L}_{q_{i+1}}}[X_{\mathcal{L}_{q_i}{}^{k_i}}/X])$ guarantees that each transition (q_i, q_{i+1}) is feasible, and 2) $\forall 0 < i \le n : X_{\mathcal{L}_{q_i}{}^0} = X_{\mathcal{L}_{q_{i-1}}{}^{k_{i-1}}}$ represents that the initial value of each path is the output

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

value of the previous path. The two parts guarantee that one trace is feasible.

Each trace $\tau \in E_{\mathcal{A}}$ represents one run of the PDA $\mathcal{A}$. The semantics of one trace $(q_0, q_1, \ldots, q_n)$ is one execution of the loop, which can be represented by the execution of paths: $(\mathcal{L}_{q_0}{}^1, \mathcal{L}_{q_1}{}^{k_1}, \ldots, \mathcal{L}_{q_{n-1}}{}^{k_{n-1}}, \mathcal{L}_{q_n}{}^1)$, where each path $\mathcal{L}_{q_i}$ in the trace will execute k_i times. Note that $\mathcal{L}_{q_0}$ and $\mathcal{L}_{q_n}$ can only be executed once since they are one-time paths. A trace contains a cycle if it contains a subsequence $(q_i, \ldots, q_j, q_i)$. The cycle $(q_i, \ldots, q_j)$ can be consecutively executed with multiple iterations in the trace.

Example 2. Fig. 1(c) and 2(c) give two PDAs. The red node represents the initial state, and the double-level node represents the accepting state. In each state, we mark the corresponding path with its possible number of executions (the symbol * in the figures means the path is an iterative path). The edge between two nodes represent the transition. In Fig. 1(c), we have the states $Q = \{q_0, q_1, q_2, q_3\}$ which correspond to the paths $\{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, the initial state q_0, the transitions $T = \{(q_0, q_1), (q_0, q_2), (q_0, q_3), (q_1, q_2), (q_2, q_1), (q_1, q_3)\}$, and the accepting states $accept = \{q_3\}$. q_0 can transit to q_1, q_2 and q_3 since σ_1, σ_2 and σ_3 can be executed after the execution of σ_0. q_1 can transit to q_2, since $x \geq z$ may be satisfiable after certain iterations of σ_1. In Fig. 1(c), there are four type of traces $E_{\mathcal{A}} = \{\tau_1 = (q_0, q_3), \tau_2 = (q_0, q_1, q_3), \tau_3 = (q_0, (q_1, q_2)^+, q_1, q_3), \tau_4 = (q_0, (q_2, q_1)^+, q_3)\}$, where $+$ means one or more executions of the sequence. $E_{\mathcal{A}}$ represents all possible executions of the loop. The details of constructing the PDA will be described in Section 4.

It is worth mentioning that our path-based PDA is generic with respect to unnested and nested loops due to the well-defined paths in the CFG. The difference is that, in unnested loops, one iterative path usually starts from the entry block and goes back to the entry block. Hence, one iteration of an unnested loop is one execution of the iterative path. In nested loops, as one iteration of the outer loop contains the loop execution of the inner loop, the path in the outer loop is "split" by the inner loop. One iteration of the outer loop is represented by multiple paths (i.e., the split paths). Intuitively, the structure of the PDA of a nested loop is usually more complex than the PDA of an unnested loop since it may contain more paths (in outer and inner loops), which lead to more transitions among the paths and more cycles.

Example 3. In Fig. 1, one iteration of the loop (executing from the basic block b to the basic block b) is one execution of σ_1 ($b \to c \to e \to b$) or σ_2 ($b \to c \to f \to b$). However, in Fig. 2, one iteration of the outer loop (executing from basic block b to the basic block b) is a sequence of execution $(\sigma_1, \sigma_2{}^*, \sigma_3)$, which contains: one execution of σ_1 ($b \to c \to e$), greater than or equal to zero execution of σ_2 ($e \to f \to e$) and one execution of σ_3 ($e \to g \to b$).

3 Loop Analysis

In this section, we first introduce the concept of loop summarization. Then, we present a classification for loops to understand the complexity of loop summarization. Finally, we present the overview of our loop analysis framework.

3.1 Loop Summarization

For one loop with PDA $\mathcal{A}$, one loop execution is represented as one trace $\tau \in E_{\mathcal{A}}$. For the trace τ, we use X and X_τ to represent the value of the variables before and after τ. X_τ can be obtained from X by updating the variables in each execution of each path $\sigma \in \tau$ (i.e., the effect after the loop execution). Loop summarization is to statically compute the constraints $\phi(X, X_\tau)$, which describe the relationship between X and X_τ. With the loop summary $\phi(X, X_\tau)$, we can compute the final values X_τ (i.e., the postcondition) from the given initial value X.

Definition 5. Given a PDA $\mathcal{A}$ and a set of variables X, the summary of a trace $\tau \in E_{\mathcal{A}}$ is the constraints denoted as $\phi(X, X_\tau)$. The *disjunctive loop summary (DLS)* of $\mathcal{A}$, denoted as $S_{\mathcal{A}}$, is $\bigvee_{\tau \in E_{\mathcal{A}}} \{\phi(X, X_\tau)\}$, i.e., the disjunction of summaries of all traces in the PDA.

To compute the DLS for a loop $\mathcal{A}$, we need to compute the summary for each trace $\tau \in E_{\mathcal{A}}$. For each trace $\tau \in E_{\mathcal{A}}$, we can summarize the effect of each state $q \in \tau$ by determining the value change in each iteration of each path $\mathcal{L}_q$ and the number of iterations of $\mathcal{L}_q$. The number of iterations of each path depends on the path condition and the value change in the path condition. If the variables in the path condition are *induction variables*, we can usually reason about the number of iterations. Summarizing a trace containing a cycle is more challenging since the execution count of each path is obtained by summing up the count in multiple executions of the cycle. However, if we can compute the execution count of the cycle and the value change in each execution of the cycle, we can also reduce the complexity.

Based on the above analysis, summarizing a multi-path loop depends on 1) the patterns of value changes in path conditions, i.e., whether the variables are *induction* or *non-induction*, and 2) the patterns of path interleaving in each trace, i.e., *acyclic execution* or *cyclic execution*. In the following, we provide a detailed explanation for the two patterns, and also present the loops we can summarize and the loops we cannot summarize.

3.2 Loop Classification

Patterns of Value Changes in Path Conditions. Given a variable x and an iterative path σ, the updated values of a variable after consecutive executions of path σ are a *sequence of numbers*. We define the *n-th term* of the variable x in the path σ is the value of x after n consecutive executions of σ. With the domain knowledge of sequences and series [20], we can calculate the *n-th term* for some sequences, e.g., if x is an Arithmetic Sequence, we can compute $x_n = x_0 + n * d$, where x_0 is the initial term, d is the *common difference* in the sequence.

According to the patterns of value changes, we classify the variables into two types.

- *Induction Variable.* A variable x is an *induction variable* (IV) in a path σ if the value of x after n consecutive executions of σ (i.e., the *n-th* term x_n) can be computed with the initial value x_0 and the variable n. Specifically, x is a *monotonic induction variable* (MIV) in a path σ if x is an IV and the value of x is strictly monotonic

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

TABLE 1: A Classification of Loops

	IV condition ($\forall$)	NIV condition ($\exists$)
Sequential	Type 1	Type 3
Periodic		
Irregular	Type 2	Type 4

increasing or decreasing, or assigned with a constant during the execution of σ.

- *Non-induction Variable.* Otherwise, if we cannot compute the n-th term after the executions of σ for the variable x, the variable is a *non-induction variable* (NIV).

Non-induction variables mainly include: 1) the value of a variable that depends on other variables and we cannot compute its n-th term (e.g., $x = x + y^2 + z^2$), 2) the value of a variable that depends on alias, content of files or function calls (e.g., $x = x + f(y)$).

Example 4. In Fig. 1(b), x is MIV in σ_1 and σ_2 since the value of x during the execution of each path is an *Arithmetic Sequence*. We can compute $x_n = x_0 + n$ in σ_1 and $x_n = x_0$ in σ_2. Similarly, z and n are also MIVs. In Fig. 2(b), m is MIV in σ_2, and we can compute $m_n = m_0 + [(2 * i_0 - 2 * j_0 - n + 1) * n]/2$.

Notice that we cannot determine all induction variables statically because it is non-trivial to infer the value of variables during loop iterations. For example, in the loop `while(i<10) x+=a[i]`, if the values in the array *a* are equal, x is indeed an IV; otherwise, x is a NIV. In our implementation, we perform a conservative static analysis and report a variable as induction variable only when we can statically compute the n-th term with the domain knowledge of mathematics (see Section 4.2).

The path condition is a conjunction of multiple atomic conditions. Each atomic condition is the form of $g(X) \sim b$, we use $e = g(X)$ as a variable for the ease of presentation. The conditions can be classified into two types:

- *IV Condition*. A condition is an IV condition if the condition is an atomic condition $e \sim b$ and e is a MIV in the path. For example, the condition $x < n$ (an atomic condition $x - n < 0$) in Fig. 1(b) is an IV condition in all three paths since $x - n$ is a MIV in each path. The strict monotonicity can be used to determine the number of executions of the path.
- *NIV Condition*. A condition is a NIV condition if e is not a MIV.

Patterns of Path Interleaving. Given a loop with the PDA $\mathcal{A}$, we classify a loop execution into three types:

- *Sequential Execution*. There is no cycle in each trace $\tau \in E_{\mathcal{A}}$.
- *Periodic Execution*. If all the cycles in every trace are regular and periodic (the period can be used to abstract the cycle), the loop execution is a periodic execution; otherwise, it is a non-periodic execution.
- *Irregular Execution*. If there are some traces that contain non-periodic cycles, the loop execution is an irregular execution. For irregular execution, we cannot determine the execution pattern of the cycle statically because the cycle is non-periodic. Hence, we cannot compute the number of executions for each path.

```
while (x<1000)
 if(x>=0 && y>0)
  x=2*x; y--;
 else if(x>0 && y<=0)
  x--; y-=2;
 else if(x<0 && y>=0)
  x++; y=3*y;
 else
   y++;
```

(a) Type 2

```
int i=0;
while(i<1000)
 if(i<100)
  i++;
 else
  int j=random();
  assume(1<=j);
  assume(j<LINT);
  i=i+j;
```

(b) Type 3

```
while(i<n)
 if(s[i]=='a')
   i++;
 else if(s[i]=='b')
   i+=2;
 else if(s[i]=='c')
   i--;
 else
   i+=3;
```

(c) Type 4

Fig. 3: Loop Classification Examples

The Proposed Loop Classification. According to the patterns of value changes in path conditions and the patterns of path interleaving, we define a loop classification, as shown in Table 1. The first row indicates that we classify a multi-path loop based on whether each atomic condition in the paths of the loop is an *IV condition* (Type 1 and 2) or there exists some *NIV conditions* (Type 3 and 4). The first column displays the criteria of path interleaving patterns, i.e., whether all the feasible executions of the loop are *Sequential* or *Periodic* (Type 1 and 3) or there exists some loop execution that can be *Irregular* (Type 2 and 4). This loop classification helps to understand the difficultness of loop summarization, i.e., what loops can be precisely summarized, what loops can be summarized with some approximation, and what loops are challenging to summarize (see Section 5 and 6).

The loops related to integer arithmetics often belong to Type 1; and the loops that traverse a data structure often belong to Type 3 and 4, because the loop iteration depends on the content of the data structure. Intuitively, loops with NIV conditions, such as the loops that are related to complex data structures, tend to have irregular execution because the path interleaving depends on non-induction variables, which have irregular value changes and often lead to non-periodic cycles among the paths.

Example 5. The loop in Fig. 1(a) belongs to Type 1 because the conditions are IV conditions and the cycle (q_2, q_1) in the PDA is a periodic cycle. Fig. 3 presents the other types of loops. Fig. 3(a) belongs to Type 2 because all the conditions are IV conditions but the cycle (the interleaving among the four paths) in the loop is not periodic. The loop in Fig. 3(b) belongs to Type 3 since its execution is sequential (i.e., from `if` branch to `else` branch) but the variable i is a non-induction variable whose value depends on the function *random()*. Fig. 3(c) belongs to Type 4 as all the conditions are NIV conditions which depend on the array *s* and lead to the irregular execution among the paths.

3.3 Loop Analysis Framework

Fig. 4 shows the workflow of Proteus. It takes a *loop* and a set of *variables of interest* as input, and it reports a *loop summary* for the variables of interest. The variables of interest are given by the client analysis that uses the loop summary. For example, if the goal is to determine the loop bound, the variables in the conditions that may jump out of a loop are of interest; and if we use the loop summary for program verification, we will need to summarize the variables relevant to the properties to be verified. In this way,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

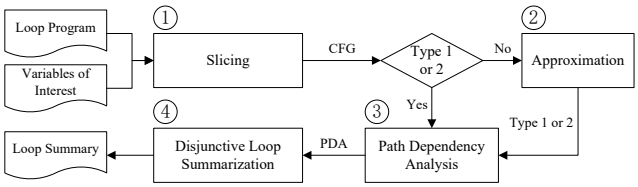


Fig. 4: An Overview of Our Framework Proteus

Proteus performs a problem-driven summarization for the loop program. Guided by the variables of interest, Proteus can further simplify the loop to generate the summary more efficiently. If the variables of interest are not specified, Proteus will try to generate a summary for all the variables in the loop.

Proteus consists of four steps to summarize a loop. Step 1 performs program slicing using the variables of interest as slicing criteria and constructs the control flow graph (CFG) for the sliced loop program.

From the CFG, we can directly determine the type of the loop from the loop conditions. If all the conditions in a loop are IV conditions, the loop belongs to Type 1 or 2; otherwise, the loop belongs to Type 3 or 4. Since summarizing NIVs is very challenging because of the uncertain value changes, we provide some approximation techniques (i.e., Step 2) to transform the loops of Type 3 and 4 to Type 1 or 2. The approximation may cause imprecise summaries but may still be useful and effective in specific applications.

In Step 3, Proteus extracts the path condition and analyzes the dependency between any two paths. We construct the PDA for the Type 1 and Type 2 loops to capture the execution order and path interleaving patterns of the paths. We can construct the PDA for Type 1 and Type 2 loops.

In Step 4, we perform a path enumeration procedure on the PDA to check the feasibility of each trace in the PDA and summarize its effect. The loop summary is a disjunction of all feasible trace summaries. The last two steps are our main contributions in this paper, which will be elaborated in Section 4 and Section 5–6 respectively.

4 PDA Construction

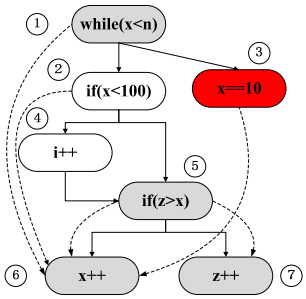
In this section, we first introduce how to perform slicing and computing the n-th term in a sequence of numbers, then we describe the construction of PDA for Type 1 and Type 2 loops.

4.1 Loop Slicing

The loop slicing removes irrelevant instructions, which leads to reduce irrelevant paths in the constructed CFG. Thus, it makes the summarization more efficient. We implement our loop slicing based on the method in [8]. The loop slicing is based on the program dependence graph (PDG) [21] that combines control flow dependencies and data flow dependencies. Taking both types of dependencies into account, we determine the relevant instructions from a *slicing criterion*. We illustrate the basic idea of determining the relevant instructions using Example 6. The algorithmic details can be found in [8].

```
1. int n=*;
2. int x=*;
3. int z=*;
4. int i=*;
5. while(x<n){
6.   if(x<100)
7.     i++;
8.   if(z>x)
9.     x++;
10. else
11.    z++;
12.}
13.assert(x==10);
```

(a) Unsliced Loop



(b) PDG

Fig. 5: Example of Loop Slicing

Example 6. Fig. 5(a) shows the unsliced loop program of Fig. 1(a); and Fig. 5(b) gives the corresponding PDG where nodes represent the instructions and edges represent the control flow dependencies (solid arrows) and data flow dependencies (dotted arrows). Here we want to check the property $x == 100$; Thus, x is a variable of the slicing criterion. In Fig. 5(b), starting from node 3, we first find and add the relevant instruction in node 1 due to the control dependency $x \geq n$, and the instruction in node 6 based on the data dependency. From node 6, we find node 1 and node 5 because x depends on the control dependencies $x < n$ and $x < z$. From node 5, we find node 7 based on the data dependency. Here, we determine node 5 does not depend on node 2 because $z > x$ can be executed under the conditions $x < 100$ and $x \geq 100$. Hence, node 2 is not added. At last, we find all the relevant nodes (marked in grey in the figure).

4.2 Computing n-th Term in a Loop Path

The n-th term [20] computation for a sequence of numbers is a classical mathematical problem and can be computed with the domain knowledge in sequences and series. Here we consider the following sequences.

- *Basic Sequence.* Arithmetic Sequence ($x_n = x_0 + c * n$), Geometric Sequence ($x_n = x_0 * c^n$) and Constant Sequence ($x_n = c$) are basic sequences, where c is a constant. We can easily compute the n-th term by the definition.
- *Dependent Sequence.* If a sequence depends on another sequence, such as $x_n = x_{n-1} + y_n$, $x_n = y_n$ or $x_n = x_{n-1} * y_n$ where y_n is a known sequence, then it is a dependent sequence, and we can compute its n-th term. For example, for $x_n = x_{n-1} + y_n$, if we can compute the sum of first n terms for y_n, then we can compute the n-th term of x_n with accumulation; i.e., calculate the sum by $(x_n - x_{n-1}) + (x_{n-1} - x_{n-2}) + \cdots + (x_1 - x_0) = x_n - x_0 = y_n + y_{n-1} + \cdots + y_1$ and get the result $x_n = x_0 + \sum_{1 \leq i \leq n} y_i$.

Based on these two kinds of sequences, we can compute the value of the variables after n executions of a path in the loop. For example, during the loop iteration, the value of x is an Arithmetic Sequence if x is updated as $x := x + c$ in one execution of a path, a Geometric Sequence if x is updated as $x := x * c$ in one execution of a path, and a Constant Sequence if x is updated as $x := c$ in one execution of a path, where c is a constant. If x is updated as $x := x + y$ in one

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

Algorithm 1: ConstructPDA

input : $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$: CFG
output: $\mathcal{A}$: PDA

1 Assume $\prod_{\mathcal{G}} := \{\sigma_1, \ldots, \sigma_n\}$;
2 $Q := \{q_0, \ldots, q_n\}$;
3 $\mathcal{L} := \{(q_0, \sigma_0), \ldots, (q_n, \sigma_n)\}$;
4 $T := \{\}$;
5 q_0 is the state, where $head(\sigma_0) = l_{pre}$;
6 $accept := \{q \in Q \mid \mathcal{L}_q \in L_e\}$;
7 **foreach** $(q_i, q_j) \in \{(q_m, q_n) \mid q_m \in Q \wedge q_n \in Q \wedge tail(\sigma_m) = head(\sigma_n) \wedge m \neq n\}$ **do**
8 Let k_{ij} be the state counter for (q_i, q_j) ;
9 $\phi_{ij} := \theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X] \wedge \theta_{\sigma_j}[X_{\sigma_i{}^{k_{ij}}}/X] \wedge$ (σ_i is an iterative path?$k_{ij} \geq 1 : k_{ij} = 1$);
10 **if** ϕ_{ij} *is satisfiable* **then**
11 $T := T \bigcup \{(q_i, q_j)\}$;
12 **return** $\mathcal{A} = (Q, \mathcal{L}, q_0, accept, T)$;

execution of a path and y is a known sequence in the path, we can compute the n-th term with dependent sequence. In Fig. 2, the value of variable m in σ_2 is a dependent sequence since we can find $m := m + (i - j)$ in σ_2 and $i - j$ is an Arithmetic Sequence. Hence, we can compute $m_n = m_0 + n * (2 * i_0 - 2 * j_0 - n + 1)/2$. Notice that the update of the variable is inferred by performing the data flow analysis on the sequence of statements in the path.

If we cannot compute the n-th term for x during the loop iteration, we regard it as NIV. Notice that, the n-th term computation is orthogonal to our loop summarization, and can be extended with more advanced algorithms.

4.3 Constructing PDA

Algorithm 1 presents the procedure of PDA construction. It takes as input the CFG of a loop, and returns the constructed PDA. At Line 1–3, we construct the states Q of PDA and the function $\mathcal{L}$ which maps each state to each path based on $\prod_{\mathcal{G}}$. For each $q_i \in Q$, we have $\mathcal{L}q_i = \sigma_i$ and $\sigma_i \in \prod_{\mathcal{G}}$. The set of transitions T is initialized at Line 4. The initial state is q_0 whose corresponding path σ_0 starts with the pre-header l_{pre} (Line 5). The accepting state is the state whose corresponding path ends with the exit block (Line 6).

Then we compute the transition between each two possible states q_i and q_j in the PDA (Line 7–11). Notice that q_i can only transit to q_j whose head equals to the tail of q_i. First, we introduce the variable $k_{ij} \geq 1$, called *state counter*, for the transition (q_i, q_j), which represents that after k_{ij} iterations of σ_i, σ_j will be executed (where $\sigma_i = \mathcal{L}_{q_i}$ and $\sigma_j = \mathcal{L}_{q_j}$) (Line 8). Then we compute the guard condition for the transition (q_i, q_j) as the conjunction of θ_{σ_i}, $\theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X]$ and $\theta_{\sigma_j}[X_{\sigma_i{}^{k_{ij}}}/X]$, which respectively represents the path condition of σ_i before executing the path σ_i, the path condition of σ_i after $k - 1$ execution of σ_i and the path condition of σ_j after k execution of σ_i (Line 9). If σ_i is an iterative path, then the iterations of σ_i can be greater than one (i.e., $k_{ij} \geq 1$); otherwise, σ_i is a one-time path and can only execute once (i.e., $k_{ij} = 1$). If ϕ_{ij} is satisfiable, q_i can transit to q_j. Then we add the transition into the set T (Line 10–11), which is proved in Theorem 1.

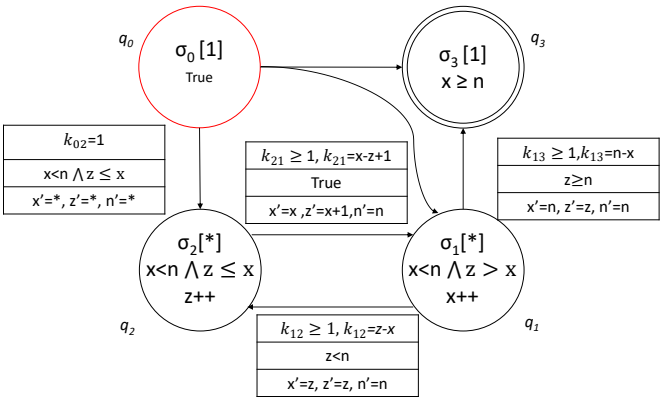


Fig. 6: Details of the PDA in Fig. 1(c)

Theorem 1. If ϕ_{ij} in Algorithm 1 is satisfiable, then q_i can transit to q_j.

Proof. (*Sketch*) Let $\sigma_i = \mathcal{L}_{q_i}$ and $\sigma_j = \mathcal{L}_{q_j}$. From Definition 3, we know if $\phi'_{ij} = \exists k_{ij} : (\bigwedge_{0 \leq m < k_{ij}} \theta_{\sigma_i}[X_{\sigma_i{}^m}/X]) \wedge \theta_{\sigma_j}[X_{\sigma_i{}^{k_{ij}}}/X]$ is satisfiable, then q_i can transit to q_j. If we can prove ϕ_{ij} can imply ϕ'_{ij} (i.e., Proposition (1)), then (q_i, q_j) is feasible.

$$\phi_{ij} \Longrightarrow \phi'_{ij} \tag{1}$$

In Algorithm 1, we know $\phi_{ij} = \theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X] \wedge \theta_{\sigma_j}[X_{\sigma_i{}^{k_{ij}}}/X]$. Note that $\theta_{\sigma_i}[X_{\sigma_i{}^0}/X]) = \theta_{\sigma_i}$. The difference between ϕ_{ij} and ϕ'_{ij} is that $\bigwedge_{0<m<k_{ij}-1} \theta_{\sigma_i}[X_{\sigma_i{}^m}/X]$ is considered in ϕ'_{ij} but not in ϕ_{ij}. Hence, to prove Proposition (1), we only need to prove Proposition (2). We assume $k_{ij} \geq 3$ in Proposition (2). When $1 \leq k_{ij} \leq 2$, it is intuitive to prove that Proposition (1) is true.

$$\theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X] \Longrightarrow \bigwedge_{0<m<k_{ij}-1} \theta_{\sigma_i}[X_{\sigma_i{}^m}/X] \tag{2}$$

We prove Proposition (2) with the proof by contradiction. We assume that $\theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X]$ is true, and $\exists\ 0 < m < k_{ij} - 1, \theta_{\sigma_i}[X_{\sigma_i{}^m}/X]$ is false. Thus, we need to prove the assumption (i.e., Proposition (3)) is a contradiction.

$$\theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{\sigma_i{}^{k_{ij}-1}}/X] \wedge 0 < m < k_{ij} - 1 \wedge \neg\theta_{\sigma_i}[X_{\sigma_i{}^m}/X] \tag{3}$$

From the definition of path condition, we know θ_{σ_i} is a conjunction of some IV conditions: $e_1 \sim 0 \wedge \cdots \wedge e_k \sim 0$, where $\sim \in \{<, \leq, >, \geq, =\}$. $\theta_{\sigma_i}[X_{\sigma_i{}^m}/X]$ is not satisfiable means $\exists 1 \leq n \leq k, e_n[X_{\sigma_i{}^m}/X] \sim 0$ is not satisfied. Thus, we derive Proposition (4):

$$\begin{aligned}(3) \Longrightarrow\ & e_n \sim 0 \wedge e_n[X_{\sigma_i{}^{k_{ij}-1}}/X] \sim 0 \\ & \wedge 0 < m < k_{ij} - 1 \wedge e_n[X_{\sigma_i{}^m}/X] \nsim 0\end{aligned} \tag{4}$$

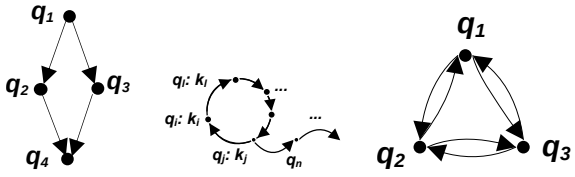
From the definition of IV condition, we know e_n is a MIV. Hence, it is monotonic increasing, decreasing or constant, which implies Proposition (5).

$$\begin{aligned}& e_n < e_n[X_{\sigma_i{}^m}/X] < e_n[X_{\sigma_i{}^{k_{ij}-1}}/X] \\ & \vee e_n > e_n[X_{\sigma_i{}^m}/X] > e_n[X_{\sigma_i{}^{k_{ij}-1}}/X] \\ & \vee e_n = e_n[X_{\sigma_i{}^m}/X] = e_n[X_{\sigma_i{}^{k_{ij}-1}}/X]\end{aligned} \tag{5}$$

For each $\sim \in \{<, \leq, >, \geq, =\}$, Proposition (5) is a contradiction with Proposition (4). Then we get a contradiction for the assumption. Thus, we can prove Proposition (1) is true; i.e., if ϕ_{ij} is satisfied, then q_i can transit to q_j. □

Based on the guard condition ϕ_{ij}, we get the following information: 1) the constraints on the state counter k_{ij} which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering



(a) Acyclic (b) Simple Cycle (c) Connected Cycles

Fig. 7: Different Structures of PDA

can represent the iteration count of σ_i, 2) the simplified guard condition by eliminating the variables k_{ij} from ϕ_{ij} with quantifier elimination, and 3) the postcondition (i.e., the value of the variables X after the transition) which can be computed based on k_{ij}.

Example 7. Fig. 6 shows a detailed PDA for the loop in Fig. 1(b). Each state is one path in the CFG with the path condition and variable update in each iteration. Each table above the transition includes: constraints about the state counter k_{ij} in the first row, the simplified guard condition in the second row, and the postcondition in the third row. For simplicity, we omit the tables from q_0 to q_1 and q_3, and they are similar to (q_0, q_2). Consider the transition (q_1, q_2). The variables are all IVs in σ_1. With Algorithm 1, we can compute $\phi_{12} = x - n < 0 \wedge z - x > 0 \wedge (x + k_{12} - 1 - n < 0) \wedge (z - x - k_{12} + 1 > 0) \wedge (x + k_{12} - n < 0) \wedge (z - x - k_{12} \leq 0)$. After the simplification of the inequalities in ϕ_{12}, we can get $z - x \leq k_{12} < z - x + 1$, i.e., $k_{12} = z - x$ (in the first row). We can further eliminate k_{12} from ϕ_{12} and get a simpilified guard condition as $z < n$ (in the second row). By substituting k_{12} with $z - x$, we can compute the postcondition (i.e., the value of the variables after the transition) as $\{x', n', z'\} = \{x + 1 \cdot k_{12}, n + 0 \cdot k_{12}, z + 0 \cdot k_{12}\} = \{z, n, z\}$ (in the third row). Similarly, we can also compute the transitions of (q_2, q_1) and (q_1, q_3).

5 Summarization For Type 1 Loops

In this section, we elaborate the algorithm for computing disjunctive loop summaries for Type 1 loops whose atomic conditions are all IV conditions.

As described in Section. 3.1, we summarize the effect for each trace, and the difficultness of summarization depends on the cycles in the traces. Fig. 7 shows different structures based on the cycles in the PDA. In Fig. 7(a), if a PDA does not contain cycles, then it is like a tree. By traversing the tree from the root node to each leaf node, we can get all the traces that are acyclic. In Fig. 7(b), if a PDA contains the cycles that are not connected (called simple cycles), we merge the cycle into one state and the PDA becomes acyclic. If the cycles are connected (called connected cycles), as shown in Fig. 7(c), then it can produce irregular execution between the cycles in each trace, which makes the summarization non-trivial.

The PDA structure of a loop depends on the number of paths and the number of transitions among the paths. For example, the PDA of the typical loop `for(i=0;i<n;i++)` belongs to the class of acyclic PDA. The PDA in Fig. 1(c) contains a simple cycle. Due to the nesting of the inner loop and the outer loop, the PDA of a nested loop contains connected cycles which include the cycles caused by the execution of the inner loop and the cycles caused by the execution of the outer loop. For example, Fig. 2(c) is a simplest PDA (there is no `if-else` branch in both of the inner loop and the outer loop), which contains the connected cycles between (q_1, q_2, q_3) and (q_1, q_3).

In the following sections, we will introduce the summarization for acyclic PDA, PDA with simple cycles and PDA with connected cycles, respectively.

5.1 Summarization for Acyclic PDA

Given the PDA $\mathcal{A}$ of one loop, we summarize the effect of the loop by callling Algorithm 2 as $SummarizeTrace(q_0, Pre(\mathcal{A}), X, S_{\mathcal{A}})$ where q_0 is the initial state, $Pre(\mathcal{A})$ is the precondition, X is a set of induction varibales that will be summarized and $S_{\mathcal{A}}$ is the result, i.e., the loop summary.

Algorithm 2 performs a path enumeration on the PDA to find each trace and summarize each transition of the trace. Its inputs include the current state q_i, the current trace condition tc under which the sub-trace from q_0 to q_i is feasible, and the values of variables X' after the previous transition. If q_i is an accepting state, the summarization for the current trace τ is finished. The value of the variables after executing the trace τ (i.e., X_τ) equals to the value of the variables after executing the path $\mathcal{L}_{q_i}$ once (i.e., $X'_{\mathcal{L}_{q_i}}$) (Line 2). Note that X_τ contains the state counters which are bounded in the trace condition tc.

On the other hand, if q_i is not an accepting state, the algorithm continues to summarize each of the transitions from q_i to its successors (Line 4–8). For each successor q_j, we update the guard condition ϕ_{ij} (computed in Algorithm 1) by substituting its variables X with X'. The constraint $tc \wedge \phi_{ij}[X'/X]$ is solved to check whether the current trace can transit to σ_j (Line 6). If feasible, the algorithm updates the current trace condition to $tc \wedge \phi_{ij}[X'/X]$ and updates the variables to X'' with k_{ij} (Line 7). Then it continues the summarization from state q_j (Line 8). Notice that the introduced variable k_{ij} represents the execution count of q_i, which is used to compute X'' after the transition from q_i to q_j (Line 7). The value of k_{ij} is bounded by the path conditions of σ_i and σ_j, i.e., in ϕ_{ij}. For example, in Example 7, the value of variable k_{12} is bounded as $z - x \leq k_{12} < z - x + 1$.

Example 8. Consider the PDA in Fig. 6 which contains one simple cycle. Here we ignore the transition (q_1, q_2) and the PDA is acyclic. Starting from the initial state q_0, it can first transit to q_3, and Algorithm 2 reaches an accepting state. Hence, the trace condition is $x \geq n$, the variables do not change, and the summary for the trace (q_0, q_3) is $x' = x \wedge z' = z \wedge n' = n$. When q_0 transits to q_1, the trace condition becomes $x < n \wedge z > x$. Consider the transition (q_1, q_3), the execution reaches an accepting state after this transition. The trace condition is updated to $x < n \wedge z > x \wedge z \geq n$ which can be simplified to $x < n \leq z$. The variables are updated to $x' = n \wedge z' = z \wedge n' = n$. Thus, the summary for the trace (q_0, q_1, q_3) is $k_{13} = n - x \wedge x' = n \wedge z' = z \wedge n' = n$.

5.2 Summarization for PDA with Simple Cycles

Summarizing the trace that contains cycles is challenging for Algorithm 2 since the execution count of the cycles is uncer-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

Algorithm 2: SummarizeTrace

input : q_i: current state, tc: current trace condition
X': updated variables, $S_\mathcal{A}$: loop summary

1 **if** $q_i \in accept$ **then**
2 $\quad S_\mathcal{A} = S_\mathcal{A} \bigvee (tc \wedge X_\tau = f_{\mathcal{L}_{q_i}}(X', 1))$;
3 **else**
4 $\quad$ **foreach** $q_j \in \{q_m \mid (q_i, q_m) \in T\}$ **do**
5 $\quad\quad$ Let ϕ_{ij} be the guard condition of (q_i, q_j);
6 $\quad\quad$ **if** $tc \wedge \phi_{ij}[X'/X]$ *is satisfiable* **then**
7 $\quad\quad\quad$ Let $X'' := f_{\mathcal{L}_{q_i}}(X', k_{ij})$;
8 $\quad\quad\quad$ $SummarizeTrace(q_j,\ tc \wedge \phi_{ij}[X'/X], X'', S_\mathcal{A})$;

Algorithm 3: HandleCyclicExecution

input : c: the cycle, tc: current trace condition
X': updated variables, $S_\mathcal{A}$: loop summary

1 Let cycle $c = (q_l, \ldots, q_j, q_i, q_l)$;
2 **if** *c is periodic* **then**
3 $\quad$ Create a new state $q_c := (\sigma_c, \theta_{\mathcal{L}_{q_c}})$;
4 $\quad$ $tran(c) := \{(q_m, q_n) | \exists q_m \in c,\ (q_m, q_n) \in T \wedge q_n \notin c\}$;
5 $\quad$ **foreach** $(q_m, q_n) \in tran(c)$ **do**
6 $\quad\quad$ Compute the transition (q_c, q_n);
7 $\quad\quad$ $X'' = f_{\mathcal{L}_{q_m}}(\ldots f_{\mathcal{L}_{q_c}}(X', k_{cn}) \ldots, k_m)$;
8 $\quad\quad$ $SummarizeTrace(q_n,\ tc \wedge \phi_{cn}[X'/X], X'', S_\mathcal{A})$;
9 **else**
10 $\quad$ Summarize other types;

tain, which may cause the algorithm not to terminate. However, if the variables are updated in the cycle regularly, we can merge the cycle as one state and reduce the complexity of PDA. In this section, we first introduce how to determine periodic cycles, then how to summarize for periodic cycles.

Determine Periodic Cycles. For a cycle $c = (q_l, q_{l+1}, \ldots, q_i)$ (q_i can transit to q_l), we regard it as a state q_c. Each execution of q_c represents one sequence of executions of the states in the cycle c. For the ease of presentation, for each $q_n \in c$, we use k_n[3] rather than $k_{n,n+1}$ to represent the state counter in the transition from q_n to its successor in the cycle. The path condition of state q_c is $\theta_{\mathcal{L}_{q_c}} = \theta_{\mathcal{L}_{q_l}} \wedge \theta_{\mathcal{L}_{q_{l+1}}}[f_{\mathcal{L}_{q_l}}(X, k_l)/X] \wedge \cdots \wedge \theta_{\mathcal{L}_{q_i}}[f_{\mathcal{L}_{q_{i-1}}}(\ldots f_{\mathcal{L}_{q_l}}(X, k_l) \ldots, k_{i-1})/X]$, which is the weakest precondition to trigger the execution of the cycle c. If $\theta_{\mathcal{L}_{q_c}}$ is satisfiable before the loop executes $\mathcal{L}_{q_l}$, the path condition of each state in the cycle will be satisfiable. Hence, the cycle c can be executed.

Here we classify the cycles into two types.

- A cycle $(q_l, q_{l+1}, \ldots, q_i)$, regarded as a state q_c, is a *periodic cycle* if the state counters $k_l, \ldots, k_i$ are IVs, and each atomic condition in $\theta_{\mathcal{L}_{q_c}}$ is an IV condition in $\mathcal{L}_{q_c}$.
- Otherwise, it is a non-periodic cycle.

Specifically, we check whether a cycle $(q_l, q_{l+1}, \ldots, q_i)$ is periodic by two steps : 1) check whether the state counters $k_l, \ldots, k_i$ are IVs, and 2) for each atomic condition $e \sim 0$ in $\theta_{\mathcal{L}_{q_c}}$, check whether e is a MIV. The constraints on the state counters $k_l, k_{l+1}, \ldots, k_i$ can be obtained from the constraints ϕ (see Line 9 in Algorighm 1). With the method in Section 4.2, we first check whether each state counter is IV. Based on the state counters, we then compute the value of the expression e after one execution of q_c for each atomic condition $e \sim b$: $e' = f_{\mathcal{L}_{q_i}}(\ldots f_{\mathcal{L}_{q_l}}(e, k_l) \ldots, k_i)$, which can be used to check whether e is a MIV.

Here the key point is to check whether the state counters in the cycle are IVs because the changes of the variables depend on the state counters. If the state counters are IVs, then the execution of the cycle has certain regularities, i.e., the execution count of each path in the cycle can be determined during the execution of the cycle. For example, for one cycle (q_1, q_2) whose state counters k_1 and k_2 are IVs, suppose they increase by one in each iteration of the cycle, then we can infer its execution is periodic: $\mathcal{L}_{q_1}, \mathcal{L}_{q_2}, \mathcal{L}_{q_1}{}^2, \mathcal{L}_{q_2}{}^2, \mathcal{L}_{q_1}{}^3, \mathcal{L}_{q_2}{}^3 \ldots$ If we cannot determine whether the state counters are IVs or whether each atomic condition is MIV, then we conclude it is an aperiodic cycle.

Summarization of Periodic Cycles. For the periodic cycle, we can merge it as a new state since we can summarize the effect of each execution of the cycle. Note that a cycle can be executed partially. For example, Fig. 8 shows the execution of Fig. 7(b), which consists of two parts: 1) k (≥ 1) executions of the complete cycle (the red part), and 2) one execution of the remaining chain $q_l, \ldots, q_j, q_n$ (the blue part). The chain is a part of the execution of the cycle, from which the loop leaves the cycle and executes other state q_n.

Algorithm 3 shows the procedure to handle a periodic cycle. Its input is the cycle $c = (q_l, \ldots, q_j, q_i, q_l)$ with the current trace condition tc and the current value of the variables X'. If c is a periodic cycle, we substitute the cycle with a new state q_c (Line 3), whose path condition is $\theta_{\mathcal{L}_{q_c}} = \theta_{\mathcal{L}_{q_l}} \wedge \theta_{\mathcal{L}_{q_{l+1}}}[f_{\mathcal{L}_{q_l}}(X, k_l)/X] \wedge \cdots \wedge \theta_{\mathcal{L}_{q_i}}[f_{\mathcal{L}_{q_{i-1}}}(\ldots f_{\mathcal{L}_{q_l}}(X, k_l) \ldots, k_{i-1})/X]$ where the constraints about the state counters $k_l, \ldots, k_j, k_i$ are from the summary in the sub-trace $(q_l, \ldots, q_j, q_i, q_l)$. For an aperiodic cycle, the summarization will be described in Section 6 (Line 10).

Then, the algorithm computes every transition (q_m, q_n) that can leave the cycle, where q_m is in the cycle, and q_n is not in the cycle and is the successor of q_c (Line 4). For each transition (q_m, q_n), it computes the dependency between q_c and q_n, i.e., the transition (q_c, q_n). The computation of the transition (q_c, q_n) (Line 6) is similar to Algorithm 1. There are two possible cases: 1) there is no remaining chain (i.e, q_n is triggered after several complete executions of q_c), and 2) there is one remaining chain $(q_l, \ldots, q_m)$ after several complete executions of q_c (see Fig. 8).

For the first case, we can directly use Algorithm 1 to compute the transition (q_c, q_n). For the second case (in Fig. 8), the last iteration of state q_c is not complete but a chain (part of the execution) before transiting to q_n. Here we consider the effect of the chain and compute the transition based on the observation: $\theta_{\mathcal{L}_{q_c}}$ is satisfiable after $k_{cn} - 1$ iterations of q_c, and $\theta_{\mathcal{L}_{q_n}}$ is satisfiable after k_{cn} iterations of q_c and the execution of chain $(q_l, \ldots, q_m)$, where k_{cn} is the state counter from q_c to q_n. Since the cycle is periodic, we can compute 1) the values of the variables X after k_{cn-1} executions of the cycle (denoted as X_1) and 2) the values of X after k_{cn} executions of the cycle and the chain (denoted as X_2). Further, we can check whether q_c can transit to q_n

3. The value of each state counter can be different during the execution of the cycle.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

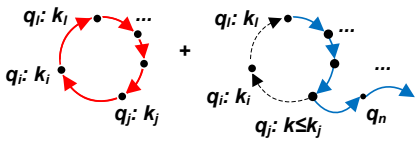


Fig. 8: Periodic Cycle Execution

TABLE 2: The Summarization Process for (q_1, q_2, q_1, q_2)

$trace$	tc	X'
$(\boldsymbol{q_1}, q_2)$	$x < n \wedge$ $z > x \wedge z < n$	$k_{12} = z\text{-}x$, $x' = z,\ z' = z,\ n' = n$
$(q_1, \boldsymbol{q_2}, q_1)$	$x < n \wedge z > x$ $\wedge z < n \wedge true$	$k_{12} = z\text{-}x,\ k_{21} = 1,$ $x' = z,\ z' = z{+}1,\ n' = n$
$(q_1, q_2, \boldsymbol{q_1}\ q_2)$	$x < n \wedge z > x$ $\wedge z < n \wedge z + 1 < n$	$k_{12} = z - x = 1,\ k_{21} = 1,$ $x' = z{+}1,\ z' = z{+}1,\ n' = n$

by checking whether $\theta_{\mathcal{L}_{q_c}} \wedge \theta_{\mathcal{L}_{q_c}}[X_1/X] \wedge \theta_{\mathcal{L}_{q_n}}[X_2/X]$ is satisfiable (Line 6).

Finally, we update the value of the induction variables X as X'' after executing the cycles and the chain based on the state counters (Line 7) and continue the summarization from the state q_n (Line 8).

Example 9. Table 2 shows the summarization process for (q_1, q_2, q_1, q_2) in Fig. 6. We can know that $k_{12} = k_{21} = 1$, which means the state counters are IVs in the cycle $c = (q_2, q_1)$. The path condition $\mathcal{L}q_c$ can be computed as $\theta_{\sigma_c} = \theta_{\mathcal{L}_{q_2}} \wedge \theta_{\sigma_1}[f_{\mathcal{L}_{q_2}}(X, 1)/X]$=$(x < n \wedge z \le x) \wedge (x < n \wedge x < z{+}1)$. If θ_{σ_c} is satisfiable, then the cycle must be executed at least once. Each atomic condition in θ_{σ_c} is an IV condition (e.g., $x < z + 1$ is IV condition since $(x - z - 1)$ is MIV in σ_c), thus c is a periodic cycle. In each execution of the cycle, we know both x and z increase by one, and n does not change. Fig. 9 shows the PDA after merging the cycle as the new state q_c. In this case, the cycle has no chain since $k_{12} = 1$. Hence we can compute the transition (q_c, q_3) with Algorithm 1, and $tran(c)$ is $\{(q_1, q_3)\}$. Finally, the trace of the loop execution is $(q_0, q_1, (q_2, q_1)^+, q_3)$, and its summary is $k_{12} = z - x \wedge k_{c3} = n - z \wedge x' = n \wedge z' = n \wedge n' = n$. Similarly, we can also compute the summary for another trace $(q_0, q_2, (q_1, q_2)^+, q_1, q_3)$ as $k_{21} = x - z + 1 \wedge k_{c1} = n - x - 1 \wedge k_{13} = 1 \wedge x' = n \wedge n' = n \wedge z' = n$. Differently, the second trace contains a chain q_1 which transits to q_3 after the cycle (q_1, q_2).

For the cycle (q_1, q_2, q_3) in Fig. 2(c), the values of the state counters k_{12} and k_{31} are always one, and the state counters k_{23} is an Arithmetic Sequence since $k_{23} = i$ and i is an Arithmetic Sequence. The path condition of the cycle is $i < n \wedge j[0/j] < i \wedge j[i/j] \ge i$ (i.e, $0 < i < n$). Based on the information, we conclude that the cycle is a periodic cycle. With the method in Section 4.2, we can summarize the n-th term of m in the cycle as $m_n = m_0 + (3 * i_0^2 * n + 3 * i_0 * n^2 + n^3 - n)/6$, where $i_0 \ge 1$.[4].

5.3 Summarization for PDA with Connected Cycles

It is non-trivial to summarize PDA with connected cycles due to the interleaving between the cycles. For example, in Fig. 7(c), one trace can execute $((q_1, q_2)^*, (q_2, q_3)^*, (q_3, q_1)^*,$ $(q_1.q_2)^*, (q_2, q_1, q_3)^*, \ldots)$. This trace not only contains the interleaving between states (e.g., q_1 and q_2), but also contains the interleaving between cycles (e.g., (q_1, q_2), (q_2, q_3) and (q_3, q_1)). The length of the trace is non-deterministic because of the interleaving between the cycles. To reduce the complexity, we can try to compute the dependencies of the cycles, which is similar with the path dependencies. If we can infer that the execution of the cycles is sequential (i.e., there is no interleaving among the cycles) under certain preconditions, then we can summarize the loop by summarizing a sequence of simple cycles in the PDA.

Actually, the PDAs of nested loops often contain connected cycles since the cycles from the outer loop and inner loop can be connected. However, many connected cycles are usually sequential (e.g., for the common rectangular loops and triangular loops [22]). Even for some complex nested loops, their connected cycles can be sequential under certain preconditions.

Algorithm 4 presents the main procedure to handle connected cycles. Given a PDA with connected cycles, we traverse it with Algorithm 2 from the state q and traverse for each trace tr (Line 1). During the traversal, if one cycle is detected, we check whether it has been summarized in the trace before (Line 3). If the cycle appears before, which indicates the interleaving among the cycles, we cannot handle such cases (Line 4). Otherwise, we summarize each simple cycle c with Algorithm 3 (Line 6) and continue the summarization from each successor of c (Line 8).

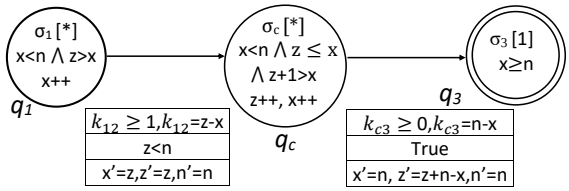


Fig. 9: The Merge of a Periodic Cycle

Example 10. In Fig. 2(c), from the structure of the PDA, we see there is one PDA with connected cycles. The cycle $c_1 = (q_1, q_2, q_3)$ and cycle $c_2 = (q_1, q_3)$ are connected. However, with the precondition of the loop $i = 1$, we find that c_2 will never be executed. Even if i can be any value before the loop, we can also find that c_2 cannot be executed after c_1. Thus, we can summarize the loop including connected cycles because the execution of the cycles is sequential.

5.4 Soundness

Theorem 2. (Soundness) Given a Type 1 loop with the PDA $\mathcal{A}$ and the summary $\bigcup_{\tau \in E_{\mathcal{A}}}\{\phi(X, X_\tau)\}$, the summarized constraints $\phi(X, X_\tau)$ for each trace τ are satisfiable after each concrete execution of τ.

Proof. (*Sketch*)Our goal is to prove that there is no such a concrete execution that follows the trace τ but whose output does not satisfy the constraints $\phi(X, X_\tau)$. To prove it, we need to prove that each concrete execution of τ satisfies the constraints $\phi(X, X_\tau)$. Without loss of generality, suppose one concrete execution follows the trace $\tau = (q_0, \ldots, q_n)$. The input value of the variables is X, and the output value of the variables after the concrete execution is X' which can

4. In the example Fig. 2(a), it can be simplified as $m_n = 1 + [n * (n + 1) * (n + 2)]/6$ with $m_0 == 1 \wedge i_0 == 1$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

Algorithm 4: SummConnectedCycle

input : q: the current state, tr: the current trace

1 Perform summarization for the trace tr from the state q with Algorithm 2;
2 **if** *cycle c is detected* **then**
3 **if** *c is in tr* **then**
4 Summarize for irregular execution;
5 **else**
6 Summarize c with Algorithm 3;
7 **foreach** *q_i as the successor of c* **do**
8 SummConnectedCycle(q_i,tr);

be regarded as a set of equations. We prove that X' implies the summarized constraints $\phi(X, X_\tau)$.

The summary $\phi(X, X_\tau)$ is computed based on the accumulation of each transition in the trace τ (Algorithm 2). For any transition (q_i, q_j), suppose the input before the transition is X_0, and the outputs of summarization and concrete execution are respectively X_s and X_c. If we can prove X_c implies X_s, then we can prove the theorem with induction in each transition of the trace.

The output of the transition is computed based on the input and the state counter, i.e., $X_s = f_{\sigma_i}(X_0, k_{ij})$ (Line 7 in Algorithm 2). Since the variables X are all IVs, if we can prove the concrete execution count of $\mathcal{L}_{q_i}$ implies the constraints about k_{ij}, then X_c implies X_s. From Theorem 1, we know if q_i can transit to q_j, it is sufficient to get that $\exists k_{ij} \geq 1 \colon \theta_{\sigma_i} \wedge (\bigwedge_{1 \leq m < k_{ij}} \theta_{\sigma_i}[X_{\sigma_i{}^m}/X]) \wedge \theta_{\sigma_j}[X_{\sigma_i{}^{k_{ij}}}/X]$ is satisfiable, which guarantees that the concrete execution (from q_i to q_j) implies the constraints about k_{ij}. Thus, we prove that the summarized result can be implied by the result of the concrete execution.

Note that the periodic cycle in the trace does not affect the soundness. The periodic cycle is transformed to one new state, and the execution of the new state (i.e, cycle) is equivalent to the execution of the states in the cycle. After the transformation, it is similar to prove that the result after the new state to its successor is also implied by the result of the concrete execution. □

5.5 Discussion about Non-Terminating Loops

We first emphasize that we handle the loops that always terminate in this paper for simplicity. Nevertheless, the algorithm can also be used to summarize the terminating traces for non-terminating loops. In this section, we discuss the summarization for non-terminating loops.

A loop is non-terminating if there are some inputs that make the loop not terminate. Hence, the PDA of a non-terminating loop can contain non-terminating traces and terminating traces for different inputs, while the PDA of a terminating loop only contains terminating traces for all inputs. For example, in the loop *while (x<0) x=x+a;*, the trace is terminating when the inputs satisfy the condition $x \geq 0 \vee a > 0$, and the trace is non-terminating when the inputs satisfy the condition $x < 0 \wedge a \leq 0$.

The approach in this paper is to compute summaries for terminating traces. For non-terminating traces, the summarization is meaningless since they have no outputs. The challenge is how to detect the non-terminating traces. Proving termination and non-termination is a typical and hard research problem, which is beyond the scope of this paper. Furthermore, we have proposed a loop termination analysis in [17] based on the PDA. Hence, we can first perform the termination analysis, and then summarize for terminating traces and report the non-terminating traces.

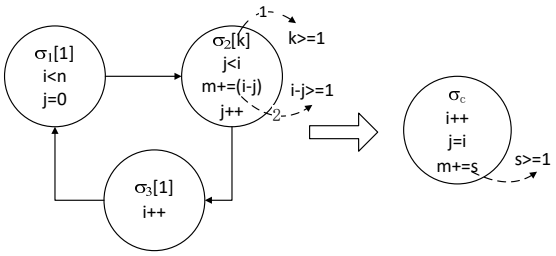


Fig. 10: Transform with Approximation

6 Summarization for Types 2, 3 and 4 Loops

It is non-trivial to precisely summarize Type 2, 3 and 4 loops as they contain NIV conditions, irregular executions or both. We introduce several approximation techniques to Proteus to facilitate the summarization, which will make an over-approximation of the summary, which may cause unsoundness in finding bugs but can be still effective for other applications, such as proving safety, termination analysis and bound analysis.

NIV Condition. The NIV conditions are difficult to summarize because of the unpredictable value change for non-induction variables. Algorithm 2 cannot summarize the loop that has non-induction variables as we cannot compute its $n\text{-}th$ term (at Line 7 in Algorithm 2) or cannot bound the range for the state counters (i.e., k) between each two states. We introduce three strategies to compute the summary as follows.

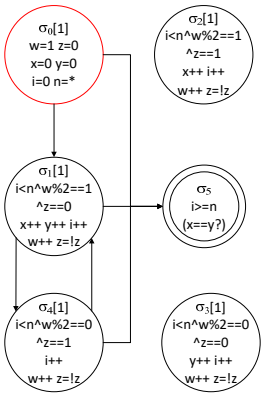
1) For non-induction variables, we apply the interval approximation to compute the summary of the variable using inequality. For example, if $x := x + c$ in each iteration of a path and c is not an IV, then x is a non-induction variable. However, if we know $1 \leq c \leq 5$, then we can approximate the computation as $x_0 + n \leq x_n \leq x_0 + 5 * n$, where x_n is the approximated $n\text{-}th$ term in the path. Similarly, in a cycle, if the state counters of some states are not IVs, then it is not a periodic cycle. If we know the interval, we also try to approximate the iteration count of the state with inequality. The summary is similar to the summary of Type 1 loops.

Example 11. Consider the cycle in Fig. 2(c), assume Algorithm 1 is limited to compute the $n\text{-}th$ term for m and the state counter k_{23} (notice that the limitation is not a theoretical limitation but in the used algorithm), thus m is a NIV and the cycle is not periodic. In Fig. 10, we show the cycle and try to summarize it with approximation. From the PDA, we can infer that $k_{23} \geq 1$ and $i - j \geq 1$. We use s to represent the approximation of $i - j$. By approximating the computation, we can transform the aperiodic cycle c to one state. In each iteration of σ_c, we know $s \geq min(k) * min(i - j) \geq 1$. At last, we can summarize $m = s * n \geq n$. The result can

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering


```
1. int w = 1, x = 0;
2. int y = 0, i = 0;
3. int n = nondeter();
4. bool z = 0;
5. while(i<n){
6.   if(w%2)
7.     x++;
8.   if(!z)
9.     y++;
10. w++;
11. z = !z;
12. i++;
13.}
14.assert(x == y);
```

(a) Loop



(b) PDA

Fig. 11: A revised version from P10 program in [19]

still be used to prove the safety. We can see the result is an over-approximation of the result in Example 9.

2) For some specific NIV conditions, such as including modulo operation or boolean variable, which can make the path executed once, we can also summarize for the loops. For example, in the path $if(bv)\ \{bv = !bv; x{+}{+}\}$ where the condition bv is a NIV condition, we can infer that the path can only be executed once, and then it will execute other paths. Hence, x is increased by one.

Example 12. Consider the loop with its PDA in Fig. 11, σ_0 is the initial state and σ_5 is the accepting state. By the dependency analysis, we infer that σ_2 and σ_3 cannot be executed under the precondition. Both of the two conditions in the if branches are NIV conditions. However, we can infer that each state can only be executed once during the transition. For example, in the transition from σ_1 to σ_4, σ_1 is executed once because of the update of w and z. Hence, the cycle (σ_1, σ_4) is a periodic cycle (in each iteration of the cycle, both of σ_1 and σ_4 can be executed once) and we can summarize the effect for the induction variables x and y with the previous algorithms. At last, we can prove that the property $x == y$ is satisfied with the summary.

3) For the NIV condition (suppose it is p) whose values are dependent on the input or context but not the loop execution, we abstract p and $\neg p$ as *true*, which is also an over-approximation. Then we need not compute its $n-th$ term. For example, in the NIV condition $a[i] > 3$, the value of $a[i]$ depends on the value of the array a and not the loop iteration, and thus we regarded $a[i] > 3$ and $a[i] <= 3$ as true since both of them can be satisfiable.

Example 13. In the loop `assume(i>0); while(i>=0 && v[i]>key) i--;`, whose PDA contains three states σ_1: $\{i >= 0 \wedge v[i] > key,\ i--\}$, σ_2: $\{i >= 0 \wedge v[i] \leq key\}$ and σ_3: $\{i < 0\}$. The conditions $v[i] > key$ and $v[i] \leq key$ are abstracted as true. Then we can summarize the two traces of the loop as: (1) the trace summary for (σ_1, σ_2) is $(i > 0,\ 1 \leq k_{12} \leq i \wedge i' = i - k_{12})$ and (2) the trace summary for (σ_1, σ_3) is $(i > 0,\ k_{13} = i + 1 \wedge i' = -1)$. Note that this loop summary is also precise here. It is unsound when the content of data structures are updated before or in the loop.

Algorithm 5: IrregSummarize

input : $\mathcal{A}$: PDA, $Pre(\mathcal{A})$: precondition
output: $S_{\mathcal{A}}$: loop summary

1 Let $\{q_1, \ldots, q_n\}$ = Q\accept;
2 Let $k_1 \geq 0, \ldots, k_n \geq 0$ be the path counters of the states in $\{q_1, \ldots, q_n\}$;
3 **foreach** $q_i \in accept$ **do**
4 **foreach** $q_j \in \{q_m \mid (q_m, q_i) \in T\}$ **do**
5 Let $X' := f_{\mathcal{L}_{q_n}}(\ldots f_{\mathcal{L}_{q_j}}(X, k_j - 1) \ldots, k_n)$;
6 Let $X'' := f_{\mathcal{L}_{q_n}}(\ldots f_{\mathcal{L}_{q_j}}(X, k_j) \ldots, k_n)$;
7 $S_{\mathcal{A}} := S_{\mathcal{A}} \bigvee\ (Pre(\mathcal{A}) \wedge \neg\theta_{\mathcal{L}_{q_i}}[X'/X] \wedge \theta_{\mathcal{L}_{q_i}}[X''/X])$;
8 **return** $S_{\mathcal{A}}$;

We will discuss the unsoundness in Section 7.

Irregular Execution. For loops with irregular path executions (e.g., the connected cycles which cannot be handled by Algorithm 4), the interleaving pattern cannot be determined. Hence, we do not know the number of the iterations for each path. In this case, we do not consider the interleaving order in any transition, but consider the total effect of each path during the whole loop execution by introducing a *path counter* [7] $k_i \geq 0$ for each path σ_i. Each path counter k_i can be used to compute the values of the IVs after k_i executions of the loop.

Here our assumption is that each variable is an IV and its sequence is Arithmetic Sequence whose output does not depend on the execution order of the paths. For example, suppose x increases by n_1 in the first path. If x increases by n_2 in the second path, we will compute the output after the two paths as $x_0 + n_1 + n_2$. However, if x multiplies by n_2 in the second path, we will compute it as $x_0 * n_2 + n_1$ or $(x_0 + n_1) * n_2$.

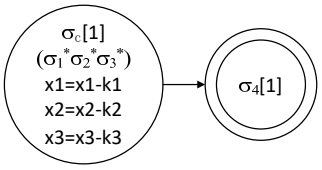
Algorithm 5 shows the procedure to summarize the loop with irregular execution. Let $\{q_1, \ldots, q_n\}$ be the states of irregular execution (Line 1). We introduce the path counters for each state (Line 2). The path counter means that the corresponding path can be executed any times. Based on the path counters, we abstract the irregular states as one state q_c. Then we compute the summary based on the transition from q_c to each accepting state (Line 3-7). Intuitively, q_c can transit to the accepting state q_i from the state q_j, which can transit to q_i (Line 4). After $k_j - 1$ executions of q_j and the executions of other states, we compute the value of the variables as X' (Line 5), and the path condition of the accepting state $\theta_{\mathcal{L}_{q_i}}$ does not hold (Line 7). After k_j executions of q_j and the executions of other states, we compute the value of the variables as X'' (Line 6), and $\theta_{\mathcal{L}_{q_i}}$ holds (Line 7).

Example 14. The loop in Fig. 12(a) has irregular executions among the states σ_1, σ_2 and σ_3. σ_4 is the accepting state. We introduce path counters k_1, k_2 and k_3 to represent their total execution count for the paths σ_1, σ_2 and σ_3. Then we merge the three states into one state σ_c, the variables after the state can be $x_1' = x_1 - k_1 \wedge x_2' = x_2 - k_2 \wedge x_3' = x_3 - k_3$. There are three possible transitions from σ_c to σ_4 since each of the three states can transit to σ_4. Consider the state σ_1 which can transit to σ_4 whose path condition is $\theta_{\sigma_4} = \neg(x_1 > 0 \wedge x_2 > 0 \wedge x_3 > 0)$, *after* $k_1 - 1$, k_2, k_3 *iterations of* σ_1, σ_2 *and* σ_3, *the*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering


```
1. while(x1>0&&x2>0&&x3>0)
2.  if (nondet())
3.     x1=x1-1;
4.  else if (nondet())
5.     x2=x2-1;
6.  else
7.     x3=x3-1;
8.  assert(x1==0||x2==0
       ||x3==0);
```

(a) Loop [13]



(b) PDA

Fig. 12: Irregular Execution

condition $\neg\theta_{\sigma_4}$ *holds; and after* k_1 *iterations of* σ_1*, the condition* θ_{σ_4} *holds*, which implies the constraints $x_1-(k_1-1)>0\wedge x_2'>0\wedge x_3'>0\wedge(\neg(x_1-k_1>0\wedge x_2'>0\wedge x_3'>0))$. By simplifying the constraints, we can get x_1'=$0\wedge x_2'>0\wedge x_3'>0$. Similarly, for other two transitions (i.e., from the state σ_2 or σ_3), we can compute x_2'=0 or x_3'=0. The summary can be used to prove the property after the loop successfully.

Theorem 3. (Termination) The summarization algorithms always terminate.

Proof. (*Sketch*) We first prove the termination of the algorithms for Type 1 loops. For the acyclic-PDA summarization, Algorithm 2 is a traversal on a tree and terminates intuitively. For the PDA with only simple cycles that are periodic, each cycle is transformed to a new state. Hence, the cyclic-PDA becomes an acyclic-PDA and it also terminates. For the PDA with connected cycles, we traverse the PDA and check whether one cycle exists before (Line 3 in Algorithm 4). If yes, we return `UNKNOWN`. Otherwise, in the connected cycles, the total number of the cycles is finite, and thus the summarization of each trace is finite since each cycle can only exist once in each trace. Thus, the algorithm always terminates.

For the other loop types, we perform the approximation for NIV conditions and irregular execution. For NIV conditions, we approximate them into IV conditions, then the loops become Type 1. The approximation does not affect the termination. For the irregular execution, Algorithm 5 contains two while loops, which always terminate since the sets $accept$ and T are finite. □

7 Evaluation

The goals of our experiments are 1) to study the distributions of loop classification in loop programs, and 2) to demonstrate the usefulness and accuracy of Proteus in practical applications.

We implemented Proteus using LLVM 3.7 [23] and SMT solver Z3 [24], and applied Proteus in three important applications: loop bound analysis, program verification, and test input generation. Note that loop classification depends on the application scenarios because different applications care about different variables, for example, some Type 3 and 4 loops can become Type 2 after removing irrelevant statements. Hence, we studied the loop classifications before the applications. The experimental results are discussed in the following sections.

TABLE 3: Loop Classification for Real-Life Loops

Projects	Total	Type1	Type3	Type4
coreutils	3349	657 (19.6%)	2259 (67.5%)	433 (12.9%)
gmp	1411	641 (45.4%)	745 (52.8%)	25 (1.8%)
pcre2	209	22 (10.5%)	151 (72.3%)	36 (17.2%)
libxml	2652	498 (18.8%)	1592 (60.0%)	562 (21.2%)
httpd	1161	69 (5.9%)	1025 (88.3%)	67 (5.8%)
Total	8782	1887 (21.5%)	5772 (65.7%)	1123 (12.8%)

7.1 Application to Loop Bound Analysis

With the disjunctive loop summary, we compute the loop bound using the constraints of the state counter, which represents the execution count of the corresponding path. Each state counter is bounded by the path conditions between each pair of states; i.e., the state counter has a upper bound. In an unnested loop, each state represents iterations of the loop. Hence, for each feasible trace, we compute the bound by adding the upper bounds of all state counters in the trace. For the cycle of the PDA, we compute the total execution count by multiplying *the total execution count in each iteration of the cycle* by *the execution count of the cycle*. The loop bound is the maximum number of the bounds of all traces. For example, we compute the bound for the trace $(q_0,q_1,(q_2,q_1)^+,q_3)$ in Example 9 as $k_{12}+2{*}k_{c3}=2n-x-z$. For a nested loop, we compute the loop bound by adding the bounds of the outer loop and the inner loop. Note that, in the PDA of a nested loop, the inner loop splits the path of the outer loop into two parts (e.g., σ_1 and σ_3 in Fig 2(c)), and we only add the bound of such a path once when computing the bound of the outer loop.

Here we selected five open source projects, including coreutils-6.10, a basic module in the GNU operating system containing the core utilities, gmp-6.0.0, an arithmetic library, pcre2-10.21, the library that implements regular expression pattern matching, libxml2-2.9.3.tar, the XML C parser and toolkit developed for the Gnome project, and httpd-2.4.18, the Apache HTTP Server Project. We selected these projects to ensure their diversity so that different forms of loops are included.

To compute the loop bound using Proteus, we are interested in knowing when the exit conditions are met. Thus, we use the exit conditions as the slicing criteria to simplify the loops as the first step. We studied the loop classifications and perform summarization on the sliced loops.

In Table 3, the programs under study are listed in the first column. Under *Total* , we list a total number of loops discovered. Under *Type1*, *Type3* and *Type4*, we list the total number of Type 1, 3, 4 loops found for each of the benchmarks. There is no column for *Type 2*, as we have not found any Type 2 loops in the five benchmarks. Theoretically Type 2 loops do exist; however, we believe that such loops are difficult to understand and maintain, and the developers typically do not write such code.

The last row of the table summarizes the results for all the benchmarks, and the 8782 loops can be classified in less than five minutes. We show that for the five projects under study, we found that 21.5% of the loops belong to Type 1, 65.7% is Type 3 and 12.8% belong to Type 4. The result of the classification depends on the category of the project. For example, gmp is an arithmetic library and many of its loops

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

TABLE 4: Loop Bound Analysis Results of Real-Life Loops

Projects	Total	Type1	Type3	Type4
coreutils	1006 (30.0%)	657 (100%)	349 (15.5%)	0
gmp	1246 (88.3%)	641 (100%)	605 (81.2%)	0
pcre2	57 (27.3%)	22 (100%)	35 (23.2%)	0
libxml	665 (25.1%)	498 (100%)	167 (9.9%)	0
httpd	122 (7.6%)	69 (100%)	53 (5.2%)	0
Total	3096 (35.3%)	1887 (100%)	1209 (21.0%)	0

belong to Type 1. Proteus is able to handle such category. For the project httpd which contains many complex data structures. Hence, the conditions in the loops are usually NIV conditions and most of the loops belong to Type 3 and Type 4. By further investigating these loops manually, we found that when a multi-path loop contains NIV conditions, the loop execution is often irregular (Type 4 loops); and when the conditions of a loop are only IV conditions, the loop execution is either sequential or periodic (Type 1 loops). There is often a correlation between NIV condition and irregular execution.

Based on these sliced loop programs, we can compute the bounds for 3096 (35.3%) loops in the projects. Table 4 provides detailed results for each benchmark. Under *Type1*, *Type3* and *Type4*, we list the percentage of the loops where we successfully compute the loop bounds. Under *Total*, we show the percentage of all the loops where we can compute the loop bounds. For the Type 1 loops which can be summarized, we can compute the bound for all of them. We can also compute the bound for some of the Type 3 loops because we use the approximation. For example, in loop `while(i<100 && node.index>0) i++`, it belongs to Type 3 because $node.index > 0$ is a NIV condition. However, we can also compute a bound by the IV condition $i < 100$. All these loops are summarized in less than 10 minutes totally.

We investigated the cases where we are not able to compute the loop bounds. We found the following reasons:

- *NIV Condition*. The loops contain non-induction variables, e.g., for conditions that contain function calls, data structures. Note that many of the loops we can handle also contain function calls, but they do not affect loop conditions and can be removed via slicing.
- *Irregular Execution*. The loops have irregular interleaving of the loop paths, and the execution order of the paths affects the bounds. For example, in the loop `while(i<n) {if(a[i]==0) i++;else i-=2;}`, the value change of i depends on the execution order of the paths. Thus, we cannot summarize it, and the bound cannot be computed by our approach.

We also tried to compare our loop bound analysis results with the current techniques [9], [10], [11]. However, their tools are currently not publicly available. Instead, we compared our approach with these techniques based on the examples used in their work. Generally, our approach has three advantages: 1) When the variable in one path is not increased or decreased by one, we can compute a more precise bound than them. For example, in the loop `while(i<n) i +=2` (suppose $i = 0$ and $n > 0$), the technique in [9] computes the bound as n while we compute the bound as $\lceil n/2 \rceil$. 2) Our approach can compute a more fine-grained bound for each trace with the disjunctive summary. 3) Our approach not only can compute the bound for the loop, but also can compute the bound for each path. This is crucial and useful in some applications. For example, in the worst case execution time (WCET) analysis [25], we can extend it to compute the whole execution time based on different path bounds easily (given the estimated execution time for each path).

In addition, we found one incorrect loop bound computed by [11] (i.e., Example 3 in Fig.4 in [11]). That loop is shown in Fig. 13(a), which contains interleaving among its multiple paths.

Assume that path σ_1 takes the `if` branch (the true branch), σ_2 takes the `else` branch (the false branch), and σ_3 is the exit path. The loop has only one execution trace $(\sigma_1, (\sigma_2, \sigma_1)^*, \sigma_2, \sigma_3)$, whose summary is $(i = 0 \wedge j = 0 \wedge 0 < m < n,\ k_{12} = m \wedge k_{c2} = n - 1 \wedge k_{23} = 1 \wedge j' = 0 \wedge i' = n \wedge m' = m \wedge n' = n)$, where k_{c2} is the state counter of the dummy state σ_c that is merged from the cycle (σ_2, σ_1). In each execution of σ_c, it executes σ_1 for m times and σ_2 once. Thus, the bound is $m + (m + 1) * (n - 1) + 1 = n \times m + n$. However, the result in [11] is $n \times m$.

In summary, using DLS, we can compute a fine-grained loop bound than the existing loop bound analysis techniques since we can compute different bounds for loop paths.

7.2 Application to Program Verification

With the disjunctive loop summary, we can prove the property (denoted as ρ) in the program. Specifically, there are two cases.

- The checked properties are after the loop. For each feasible trace τ, we check whether $\phi(X, X_\tau) \wedge \neg\rho$ is satisfiable. If there exists one trace that makes $\neg\rho$ satisfiable, a counter-example is found; otherwise, the property ρ is proved.
- The checked properties are inside the loop. The property inside the loop can be regarded as an `if-else` statement that can lead to branch ρ and branch $\neg\rho$. The property is changed as $if(\neg\rho)\ err()$. Hence there are some states including the branch $\neg\rho$ in the PDA. The problem is transformed to find a feasible trace (in Algorithm 2) that reaches the state containing the branch $\neg\rho$. If we can find it, a counter-example is found; otherwise, the branch $\neg\rho$ is unreachable and the property is proved. Specially, we can extend Algorithm 2 by checking whether q_j contains $err()$ between Line 6 and Line 7. If q_j contains $err()$, then the property is not satisfied. If $err()$ is not reachable after the summarization, then the property is satisfied. The approach is also general for the properties inside non-terminating loops.

In general, both cases can be regarded as the reachability problem in the PDA. The difference is that when the property is after the loop, we check whether $\neg\rho$ is reachable from the accepting states; and when the property is inside the loop, we check whether $\neg\rho$ is reachable from any non-accepting state in the PDA.

We used the benchmark *Loops* in Competition on Software Verification 2016 (SV-COMP'16) [13] and the benchmark in [19]. The programs in these benchmarks are small

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

TABLE 5: Program Verification Results of Benchmark Loops When Comparing Proteus with the state-of-the-art tools

Benchmark	LNum	Type1	Type2	Type3	Type4	PNum	Summ	Proteus		CPAchecker		SMACK+Corral		SeaHorn	
								C	T(s)	C	T(s)	C	T(s)	C	T(s)
loops	136	69	9	35	23	64	40	37	24	32	2073	37	5211	31	19
loopacc	35	33	0	2	0	35	29	29	18	19	9205	17	14183	15	6683
looplit	18	14	2	1	1	16	14	14	8	13	208	13	7053	13	768
loopnew	8	7	0	1	0	8	8	8	4	4	1827	6	660	5	1712
prog. [19]	51	34	16	1	0	46	45	45	10	36	8152	43	989	33	1814
Total	248	157(63%)	27(11%)	40(16%)	24(10%)	169	136	133	64	104	21465	106	28096	97	10996

```
1  assume (0<m<n);
2  i := 0; j := 0;
3  while (i<n && nondet)
4   if (j<m) j++;
5   else {j := 0; i++};
```

(a)

```
1  int SIZE=*+1,a[SIZE],j=0;
2  a[SIZE/2]=3;
3  while (j<SIZE && a[j]!=3)
4    j++;
5  assert(j<SIZE);
```

(b)

Fig. 13: Loop Examples for Evaluation

but contain non-trivial loops. Note that the *loop-inv* category in *Loops* contains many assertions that are not relevant to loops, and thus we used the other four categories.

We compared our verification results with several tools: SMACK+Corral [12], CPAchecker-LPI [26] and SeaHorn [27] are the top tools with respect to correct rate in SV-COMP'16 (CPAchecker-LPI achieved the best score in SV-COMP'16 for *Loops*). Note that we selected the tools based on their correct rate rather than their score in SV-COMP'16 since we compare the number of correctly verified loop programs. The tools are configured in the same as in the competition [13]. All of them were configured with a timeout of 15 minutes.

Table 5 shows the verification results of those techniques together with the loop classification and summarization statistics. Column *Benchmark* shows the involved loop categories. Columns *LNum* and *Type1*, *Type2*, *Type3*, *Type4* respectively list the total number of the loops and the loop classification for each benchmark. Column *PNum* and *Summ* list the number of programs and the number of loops that can be summarized, respectively. Columns *C* reports the number of programs that can be correctly verified by the techniques, while columns *T* lists their time overhead. Here we only compared the programs whose loops can be summarized to show that the loop summary is complement to these tools to improve their capability on loops.

In the five benchmarks, there are totally 248 loops in 169 programs. Among them, 63% loops belong to Type 1, 11% loops belong to Type 2, 16% loops belong to Type 3 and 10% loops belong to Type 4. There are some Type 2 loops because we abstracted the conditions containing *unknown()* and *VERIFIER_nondet_int()* as true. Hence, variables in the paths are IVs but the interleaving between the paths is irregular. The classification also shows most of them are Type 1 loops which can be summarized by Proteus. For other types of loops, we perform the approximation to summarize it.

Totally, Proteus can summarize 136 (80.5%) programs. The programs in SV-COMP'16 [13] mainly contain unnested loops while the benchmark [19] contains 12 programs with nested loops. Even for these 12 programs, Proteus can summarize and verify the property correctly. With DLS, Proteus can correctly verify 133 (97.8%) loops within 64 seconds. Note that in Table 5, the time reported for Proteus includes both the time for computing DLS and the time for proving properties with DLS. Compared with other tools, 104 (76.5%) loops can be correctly verified in 21456 seconds for CPAChecker. SMACK+Corral can correctly verify 106 (77.9%) loops in 28096 seconds. SeaHorn takes 10996 seconds to correctly verify 64 (71.3%) loops. The time overhead of CPAchecker, SMACK+Corral and SeaHorn is very high because CPAchecker has 23 timeouts, SMACK+Corral has 28 timeouts and SeaHorn has 10 timeouts. The results indicate that our technique slightly outperforms these top tools on effectiveness, and significantly outperforms them on performance.

In the category *loops*, there are 40 programs we can handle. However, we obtained incorrect verification results for three programs, $linear\_sea.ch\_true\text{-}unreach\text{-}call.c$, $linear\_search\_false\text{-}unreach\text{-}call.c$ and $string\_true\text{-}unreach\text{-}call.c$. For $linear\_sea.ch\_true\text{-}unreach\text{-}call.c$, the incorrect verification result is caused by the unsound summaries with approximation. The program in Fig. 13(b) is a simplified version of $linear\_sea.ch\_true\text{-}unreach\text{-}call.c$. Our technique approximates the condition $a[j]$!=3 as $true$ and finds a counter-example *j*==*SIZE*. Actually, the content of array a is changed at Line 2, which makes the loop exit before *j*==*SIZE*. Thus, the property *j*<*SIZE* is always correct. For the other two programs, our result seems to be correct since Proteus reports $true$ for $string\_true\text{-}unreach\text{-}call.c$ and $false$ for $linear\_search\_false\text{-}unreach\text{-}call.c$. However, for $linear\_search\_false\text{-}unreach\text{-}call.c$, the counter-example we found is not the correct one but is caused from the over-approximation. For $string\_true\text{-}unreach\text{-}call.c$, we got the correct result since the assertion $(found == 0 \mid\mid found == 1)$ is always $true$, where $found$ is a boolean variable.

In summary, using DLS, Proteus can correctly verify more programs with less time overhead than existing tools for those unnested and nested loops that we can summarize. Therefore, our loop summary can be an effective complement to the existing tools.

7.3 Application to Test Case Generation

In this experiment, we show the effectiveness of loop summary in test case generation. We compared the performance of our technique with the symbolic execution tools KLEE [14] and Pex [15] using some of the programs from *loop-acc* and [19], which contain deep loops (with large loop iterations). For each program, we added an `if` branch after or in the loop and generate a test case which can reach the `if` branch with KLEE [14], Pex [15] and Proteus. Consider the condition in the `if` branch as a checked property, then the detail about generating the test case with DLS

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering


```
1  int main(void) {
2    int x = __VERIFIER_nondet_uint();
3    int y = x + 1;
4    while (x < 1024) {
5      x++;
6      y++;
7    }
8    //__VERIFIER_assert(x == y);
9    if(x != y)
10     assert(0);
11 }
```

Fig. 14: The Loop in multivar_false-unreach-call1.c

is similar to program verification (see the first paragraph in Section 7.2). Notice that our goal is not to compare the tools but to show DLS can be potentially used to scale symbolic execution.

Table 6 shows the results for seven programs. The first four programs are from *loop-acc*, and the condition in the `if` branch we added is from the assertion in the program. For example, Fig. 14 shows a program which contains the assertion statement at Line 8. We replace the statement with the `if` branch at Line 9-10 and try to generate a test case which can reach the `if` branch. The other three programs are from the benchmark [19] and contain some nonterminating functions. We manually modified the nonterminating functions and added the `if` conditions that need deep iterations. Among them, P6.c and P10.c are nested loops. In the selected programs, the loops in P06.c and P10.c are Type 3 and the loops in other programs belong to Type 1. All the modified programs can be found in the supplemental material. In the table, T/O represents that the tools cannot generate a test case within 30 minutes and times out; and F means that Pex fails to generate a test case and throws an "out of memory" exception for the large branches.

The results show that even for the simple loops, KLEE timed out for three programs and took much more time for three programs. Pex failed to generate test cases for four programs and timed out for one program. This is because symbolic execution consumes much time to keep unfolding the deep loop. On the contrary, Proteus generated test cases for all the programs in less than one second.

In summary, the state-of-the-art symbolic execution tools KLEE and Pex can take much time or throw exceptions when a loop has many iterations. In such cases, DLS can be helpful to improve the performance of these tools by utilizing the summary during symbolic execution. It also shows that the loop summary can be potentially used to find the corner case, which usually involves many iterations of a loop and is difficult to detect. We will leave the future work to integrate the loop summarization into the existing symbolic execution tools.

8 Related Work

Loop invariants are the properties that hold at each loop iteration. Loop summarization focuses on capturing the relations of variables at the entry of the loop and at the exit of the loop, and can generate symbolic constraints which hold at the exit of the loop. In this section, we first introduce the related work in these two areas, and then discuss the loop analysis in different applications as well as the relevant work in control-flow refinement.

8.1 Loop Invariant Detection

A number of advances have been made on loop invariant inference [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]. Most of them are based on abstract interpretation [4], which iterates the loop until a fixpoint is reached. To ensure the termination, they often use the *widening* operator, which can lead to imprecision. Techniques [32], [39], [40], [41], [42] are proposed to accelerate the iteration and reduce imprecision. These approaches mainly focus on conjunctive invariants, which cannot represent disjunctive program properties.

Several attempts have also been made to infer disjunctive invariants. The techniques in [33], [34] propose a static analysis to separate a loop into multiple phases and compute more precise invariants. The technique in [35] decomposes a multi-path loop into several single loops and computes invariants for each loop. However, these techniques cannot handle the loop (e.g., Fig. 1) whose PDA contains cycles because the interleaving in the cycle may make the decomposition of phases or single loops infinite. The technique in [36] uses dynamic analysis to generate disjunctive invariants over program trace points with algebras, e.g., min-plus and max-plus. Comparing with our approach, it is more expensive to compute a convex hull with generating trace points, and it generates more approximations than ours, especially for Type 1 loops. The template-based technique [43] needs user-provided templates, and thus is not fully automatic. In [44], a learning algorithm based on counter-examples is proposed to synthesize the invariants while the technique in [45] synthesizes invariants with a template-based learning method.

Compared with loop invariants, Proteus considers the relationship among paths, and computes disjunctive summary by summarizing the effect for each path. Invariants inference, especially stronger invariants inference, is more expensive than Proteus since it depends on the multiple iterations of a loop, which has been demonstrated in our experimental study. Besides, Proteus can compute symbolic values for variables after a loop or one path, thus can be directly used in more applications such as symbolic execution.

8.2 Loop Summarization

Several techniques have been proposed to summarize the effects of loops [2], [5], [6], [7], [46], [47]. LESE [2] introduces a symbolic variable *trip count* as the number of times a loop executes and uses it to infer the loop effect. The technique in [5] detects loops and *induction variables* on the fly and infers simple partial loop invariants and generates pre- and post-conditions as loop summaries. However, both of *trip count* and *induction variable* can only describe the update in the loop iteration and not the update in multiple paths. The technique in [46] also focuses on the loop with one path and each variable being changed with a constant. Also, it can handle some simple array variables with quantified formulas. Compared with Proteus, these techniques mainly focus on single-path loops and their induction variables are only the style $x := x + c$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

TABLE 6: Test Case Generation Results When Comparing Proteus with Symbolic Execution Tools KLEE and Pex

Tool	functions_false.c	phases_false.c	multivar_false.c	simple_false1.c	P06.c	P10.c	P24.c
KLEE	23 min	T/O	11.97 s	22 min	7 min	T/O	T/O
Pex	F	F	0.5 s	F	F	4.10 s	T/O
Proteus	0.06 s	0.18 s	0.05 s	0.03 s	0.48 s	0.26 s	0.31 s

APC [6] introduces *path counter* for each path to describe the overall effect of variable changes in the loop. It summarizes a loop by computing the necessary condition on loop conditions. S-Looper [7] summarizes multi-path string loops using path counters. It extracts the string pattern from each path and then generates the string constraints. The technique in [48] transforms the program by adding a path to summarize the effect of a parametric number of iterations of a loop to accelerate the detection of deep bugs. Its result is an under-approximation which can be used to detect bugs, but cannot prove properties. As an extension of [48], the technique [47] presents trace automata to reduce the executions and perform loop acceleration. However, APC, S-Looper and trace automata cannot handle loops with complex path interleaving, e.g., the loop in Fig. 1(a).

Proteus aims to model path interleaving of a multi-path loop, and computes a fine-grained DLS, which cannot be computed by any of the existing techniques. In addition, the techniques [2], [7], [47] cannot summarize the variables which are updated in nested loops. APC and APLS can handle some of the nested loops but their result may cause unsoundness. Furthermore, Proteus can summarize more variables. For example, in APLS, the *induction variable* has two limitations: 1) IV can only be modified by a nonzero constant and 2) the change is the same in all the paths. In Proteus, IV is the variable whose n^{th} term can be calculated in each path, and the change can be different in different paths. The techniques above cannot support and summarize the IVs in this paper.

PIPS [49] computes the transitive closure of the transformer that is similar to our summary. The iterative refinement process is based on discrete differentiation and integration but not abstract interpretation. However, it does not always converge and may lead to a precision loss due to magnitude overflows. Loopfrog [1] computes the symbolic abstract transformers, which does not depend on the expensive iterative fixpoint computation. However, it needs the candidate invariants, which is an art and usually abstract interpretation is used. Compared with Proteus, these techniques [1], [49] are more generic, while Proteus performs more specific algorithms for different types of loops. Hence, Proteus can compute more precise results for specific loops such as Type 1 loops. Besides, Proteus does not rely on the iterative refinement process and invariants.

Some abstract acceleration techniques [32], [50] are proposed to compute a more precise abstract effect of a loop without widening if possible. Otherwise, they use standard abstract interpretations. These techniques are limited in the computation of the eigenvalues of the transformation matrix, which is a challenging problem; and they can only accelerate the inner loops of nested loops. For example, for the variable m in the nested loop in Fig 2(a), all the techniques above cannot compute such a precise summary that is computed by Proteus.

8.3 Loop Analysis for Different Applications

Loop analysis can be applied to various domains. Here we focus our discussion on loop bound analysis and program verification.

Loop Bound Analysis. Lokuciejewski et al. [8] compute the loop iteration counts based on abstract interpretation [4]. Their polytope-based approach assumes that the variable in the loop exit condition must increase in each loop iteration. Gulawani et al. [51] compute bounds for multi-path loop based on user annotations. Gulwani et al. [10] use counter instrumentation strategies and a linear arithmetic invariant generation tool to compute the bound. However, it is limited for multi-path loops when disjunctive invariants are needed. It also fails to compute the bound for the loop in Fig. 1(a).

Gulwani et al. [11] use control-flow refinement and progress invariants to estimate loop bounds. Its bound computation relies on a standard invariant generator and the result is usually inequalities. Gulwani et al. [9] also propose a two-step solution (computing disjunctive invariants and a proof-rule based technique) to compute the bound. However, if the variables are not increased (or decreased) by one in each iteration, their result is an upper bound and not precise. Differently, Proteus can compute a precise bound with the disjunctive summarization on the PDA.

Program Verification. Bound Model Checking (BMC) is a technique to check the properties with bounded iterations of loops (e.g., [52], [53], [54]). It is mainly used to find property violations based on SMT solvers such as [24], [55], [56], but it can not prove safety properties soundly. Kroening et al. [47] overcome this problem by introducing *trace automata* to eliminate redundant executions after performing loop acceleration, which is limited for multi-path loops whose accelerated paths interleave with each other. The techniques in [57], [58] combine predicate analysis with counterexample-guided abstraction refinement. However, it depends on the discovered predicates, which are often difficult to control.

Several techniques propose to handle loops by combining BMC with k-induction. SCRATCH [59] supports combined-case k-induction [60] but needs to set k manually. However, split-case k-induction [61], [62] can change k iteratively. ECBMC [61] assigns non-deterministic values to loop termination condition variables, making induction hypothesis too weak and unsound. PKIND [63], CPAchecker [62] and KIKI [64] strengthen the induction hypothesis with auxiliary invariants. However, their effectiveness and performance depend on the inferred invariants. k-induction technique may consume much more time to get a better k. Proteus can help verify programs with loop summaries effectively, as shown in our experimental results.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2788018, IEEE Transactions on Software Engineering

In [65], the authors propose a framework to optimize the monitoring of the loops related to a property. Their goal is to find one shorter trace that is stuttering equivalent with the complete trace (obtained through loop unrolling) for checking the property. Thus, the cost of the monitoring is reduced. Differently, loop summarization is to compute the constraints, which represent the semantics of a loop and can be used to check the property statically.

Techniques such as [66], [67] check safety properties on flat counter automata (FCA) [68], where the guard condition in the transition is limited to two variables. Thus, some loops can be modeled by PDA but not by FCA. FAST [69] is a tool to perform the analysis on the counter systems based on symbolic representation and can handle unbounded integer variables. Proteus can generate more fine-grained summary than them by summarizing the effect of each trace, and can be easily extended to support applications such as bound analysis, the worst execution time analysis and testing.

Symbolic Execution. Symbolic execution [14], [15] symbolically performs program operations on variables. When symbolic execution reaches a branch, it performs a fork, and creates a new branch in each iteration of a loop. If the loop iteration count cannot be determined, it may keep unfolding the loop without covering any new branches, and fail to achieve a high coverage and detect bugs [3]. The techniques [2], [5], [6], [7] are proposed to improve the symbolic execution, and they are compared in Section 8.2. By replacing the loops with the summarized constraints, we can make the program loop-free and improve the symbolic execution very well.

8.4 Control-Flow Refinement

Several techniques [11], [35], [70], [71] are proposed to perform control flow refinement for precise program analysis. Our approach is similar to the approaches [11], [70]. They extract the paths from the control flow graph and the paths can interleave in an arbitrary manner. Then the refinement is constructed "bottom-up" in [11] and "top-down" in [70] to refine the feasible interleaving. PDA is similar to these two approaches. The differences include: 1) the path is more specific than them in nested loops. In PDA, we split the path from the outer loop to the inner loop into two paths: the path from the outer loop to the header block of the inner loop and the path in the inner loop. Consequently, PDA is general for nested loops and unnested loops which is discussed in Section 2.2. Whereas, the inner loop is refined first before refining the outer loop in [11], [70]. 2) the approach of refinement is different. The abstract interpretation and invariants are used in the refinement [11], [70], which can be expensive as it needs to iterate a loop for many times while Proteus does not. The approach [35] decomposes a loop into multiple loops to compute the disjunctive loop invariants. The approach [71] uses abstract interpretation to infer the infeasible paths, and it helps minimize the impact of the join operation. Our approach can also be used to check the feasibility of the path by checking the satisfiability of the path condition.

9 Conclusion and Future Work

In this paper, we propose the path dependency automaton (PDA) to model the dependencies between each two paths of a multi-path loop. Based on the PDA, we propose a classification for multi-path loops to understand the difficulty of loop summarization as well as a loop analysis framework Proteus to compute disjunctive loop summary for different types of loops. To the best of our knowledge, this is the first work that can identify different execution patterns of multi-path loops, and compute disjunctive loop summary in multi-path loops with complex path interleaving. Based on the PDA and loop summarization, we also perform the loop termination analysis by checking whether each trace always ends with one accepting state [17].

In the future, we plan to continue this line of research in three aspects: 1) we plan to extend the PDA for recursive function summarization. Actually, PDA can be easily extended for modeling functions by defining paths in functions. For recursive function summarization, we can handle the tail recursion since they can be transformed to loops directly. For others, they can be transformed to loops by adding the stack. Hence, we can extend PDA by introducing stack (similar with pushdown automaton [72]). 2) we hope to propose a systematic summarization for Type 2, 3 and 4 loops. In Section 6, we handle non-induction variables which depends on other variables by adopting some heuristics. We plan to develop a more systematic approach through the combination of techniques such as approximation, refinement techniques [73], [74] and machine learning. For other non-induction variables such as those depending on alias, function call or content of files, which we do not touch in the paper, we try to handle them by using alias analysis, function summarization and environment modeling. Furthermore, array is also a special type but widely used variable and we will try to summarize it with quantified formulas. 3) we plan to apply our loop summarization to software engineering and security tasks, such as vulnerability detection, compiler optimization and program debugging.

References

[1] K. Daniel, S. Natasha, T. Stefano, T. Aliaksei, and W. C. M., "Loop summarization using state and transition invariants," *Form. Methods Syst. Des.*, pp. 221–261, 2013.
[2] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *ISSTA*, 2009, pp. 225–236.
[3] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *ASE*, 2013, pp. 246 – 256.
[4] P. Couso and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.
[5] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *ISSTA*, 2011, pp. 23–33.
[6] J. Strejček and M. Trtík, "Abstracting path conditions," in *ISSTA*, 2012, pp. 155–165.
[7] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen, "S-looper: Automatic summarization for multipath string loops," in *ISSTA*, 2015, pp. 188–198.
[8] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract," in *CGO*, 2009.
[9] S. Gulwani and F. Zuleger, "The reachability-bound problem," in *PLDI*, 2010, pp. 292–304.

[10] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: Precise and efficient static estimation of program computational complexity," in *POPL*, 2009, pp. 127–139.

[11] S. Gulwani, S. Jain, and E. Koskinen, "Control-flow refinement and progress invariants for bound analysis," in *PLDI*, 2009, pp. 375–385.

[12] A. Haran, M. Carter, M. Emmi, A. Lal, S. Qadeer, and Z. Rakamarica, "Smack+corral: A modular verifier," in *TACAS*, 2015, pp. 451–454.

[13] "Competition on software verification 2016," http://sv-comp.sosy-lab.org/2016.

[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[15] N. Tillmann and J. de Halleux, "Pex-white box test generation for .NET," in *NDSS*, 2008, pp. 134–153.

[16] X. Xie, B. Chen, L. Yang, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *FSE*, 2016, pp. 61–72.

[17] X. Xie, B. Chen, L. Zou, S.-W. Lin, L. Yang, and X. Li, "Loopster: Static loop termination analysis," in *ESEC/FSE*, 2017, pp. 84–94.

[18] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, pp. 576–580, 1969.

[19] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *OOPSLA*, 2013, pp. 443–456.

[20] W. F. Trenchh, *Introduction to Real Analysis*. Library of Congress Cataloging-in-Publication Data, 2003.

[21] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *SDE*, 1984, pp. 177–184.

[22] D.-H. Chu and J. Jaffar, "Symbolic simulation on complicated loops for wcet path analysis," in *EMSOFT*, 2011, pp. 319–328.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–88.

[24] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.

[25] R. Wilhelm, E. Jakob, E. Andreas, H. Niklas, T. Stephan, W. David, B. Guillem, F. Christian, H. Reinhold, M. Tulika, M. Frank, P. Isabelle, P. Peter, S. Jan, and S. Per, "The worst-case execution-time problem – Overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[26] "Lpi," http://lpi.forge.imag.fr/.

[27] A. Gurfinkel, T. Kahsai, and J. A. Navas, "Seahorn: A framework for verifying c programs(competition contribution)," in *TACAS*, 2015, pp. 447–450.

[28] "The interproc analyzer," http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html.

[29] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL*, 1978, pp. 84–96.

[30] M. Antoine, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, pp. 31–100, 2006.

[31] M. Karr, "Affine relationships among variables of a program," *Acta Informatica*, vol. 6, pp. 133–151, 1976.

[32] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," in *POPL*, 2014, pp. 529–540.

[33] D. Gopan and T. Reps, "Lookahead widening," in *CAV*, 2006, pp. 452–466.

[34] D. Gopan and T. Reps, "Guided static analysis," in *SAS*, 2007, pp. 349–365.

[35] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *CAV*, 2011, pp. 703–719.

[36] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *ICSE*, 2014, pp. 608–619.

[37] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *CAV*, 2009, pp. 634–640.

[38] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as learning geometric concepts," in *SAS*, 2013, pp. 388–411.

[39] X. Rival and L. Mauborgne, "The trace partitioning abstract domain," in *ACM Trans. Program. Lang. Syst.*, 2005.

[40] D. Monniaux, "Automatic modular abstractions for linear constraints," in *POPL*, 2009, pp. 140–151.

[41] D. Monniaux and P. Schrammel, "Speeding up logico-numerical strategy iteration," in *SAS*, 2011, pp. 253–267.

[42] P. Schrammel and B. Jeannet, "Logico-numerical abstract acceleration and application to the verification of data-flow programs," in *SAS*, 2011, pp. 233–248.

[43] B. S. Gulavani and S. K. Rajamani, "Counterexample driven refinement for abstractinterpretation," in *TACAS*, 2006, pp. 474–488.

[44] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV*, 2014, pp. 69–87.

[45] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi, "Automatically inferring quantified loop invariants by algorithmic learning from simple templates," in *APLAS*, 2010, pp. 328–343.

[46] M. N. Seghir, "A lightweight approach for loop summarization," in *ATVA*, 2011, pp. 351–365.

[47] D. Kroening, M. Lewis, and G. Weissenbacher, "Proving safety with trace automata and bounded model checking," in *FM*, 2015, pp. 325–341.

[48] D. Kroening, M. Lewis, and G. Weissenbacher, "Under-approximating loops in c programs for fast counterexample detection," in *CAV*, 2013, pp. 381–396.

[49] C. Ancourt, F. Coelho, and F. Irigoin, "A modular static analysis approach to affine loop invariants detection," *ENTCS*, pp. 3–16, 2010.

[50] L. Gonnord and P. Schrammel, "Abstract acceleration in linear relation analysis," *Sci. Comput. Program.*, pp. 125–153, 2014.

[51] B. S. Gulavani and S. Gulwani, "A numerical abstract domain based on expression abstraction and max operator with application in timing analysis," in *CAV*, 2008, pp. 370–384.

[52] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," in *VSTTE*, 2012, pp. 146–161.

[53] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *TACAS*, 2004, pp. 168–176.

[54] P. G. de Aledo and P. Sanchez, "Framework for embedded system verification(competition contribution)," in *TACAS*, 2015, pp. 429–431.

[55] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays." in *TACAS*, 2009, pp. 174,177.

[56] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007, pp. 519–531.

[57] D. Beyer, M. E. Keremoglu, and P. Wendler, "Predicate abstraction with adjustable-block encoding," in *FMCAD*, 2010, pp. 189–197.

[58] D. Beyer and S. Löwe, "Explicit-state software model checking based on cegar and interpolation," in *FASE*, 2013, pp. 146–162.

[59] A. F. Donaldson, D. Kroening, and P. Rümmer, "Automatic analysis of dma races using model checking and k-induction," in *FMSD*, 2011, pp. 83–113.

[60] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *SAS*, 2011, pp. 351–368.

[61] J. Morse, L. Cordeiro, D. Nicole1, and B. Fischer, "Handling unbounded loops with esbmc 1.20(competition contribution)," in *TACAS*, 2013, pp. 619–622.

[62] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *CAV*, 2015, pp. 622–640.

[63] T. Kahsai and C. Tinelli, "Pkind: A parallel k-induction based model checker," in *PDMC*, 2011, pp. 55–62.

[64] M. Brain, S. Joshi, D. Kroening, and P. Schrammel, "Safety verification and refutation by k-invariants and k-induction," in *SAS*, 2015, pp. 145–161.

[65] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010, pp. 270–285.

[66] M. Bozga, R. Iosif, and F. Konecny, "Safety problems are np-complete for flat integer programs with octagonal loops," in *VMCAI*, 2014, pp. 242–261.

[67] S. Demri, A. K. Dhar, and A. Sangnier, "On the complexity of verifying regular properties on flat counter systems," in *ICALP*, 2013, pp. 162–173.

[68] M. Bozga, R. Iosif, and Y. Lakhnech, "Flat parametric counter automata," in *ICALP*, 2006, pp. 577–588.

[69] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci, "Fast: Fast acceleration of symbolic transition systems," in *CAV*, 2003, pp. 118–121.

[70] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Refining the control structure of loops using static analysis," in *EMSOFT*, 2009, pp. 49–58.

[71] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta, "Slr: Path-sensitive analysis through infeasible-path

detection and syntactic language refinement," in *SAS*, 2008, pp. 238–254.

[72] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer, "Alternating pushdown automata," pp. 92–106, 1978.

[73] A. A. Loginov, "Refinement-based program verification via three-valued-logic analysis," Ph.D. dissertation, Madison, WI, USA, 2006.

[74] S. Ray and R. Sumners, "Specification and verification of concurrent programs through refinements," *Journal of Automated Reasoning*, pp. 241–280, 2013.