11-2016

Proteus: Computing disjunctive loop summary via path dependency analysis

Xiaofei XIE
Singapore Management University, xfxie@smu.edu.sg

Bihuan CHEN

Yang LIU

Wei LE

Xiaohong LI

Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis

Xiaofei Xie[1] Bihuan Chen[2] Yang Liu[2] Wei Le[3] Xiaohong Li[1]*
[1] Tianjin Key Laboratory of Advanced Networking, Tianjin University, China
{xiexiaofei, xiaohongli}@tju.edu.cn
[2]Nanyang Technological University, Singapore {bhchen,yangliu}@ntu.edu.sg
[3]Iowa State University, USA weile@iastate.edu

ABSTRACT

Loops are challenging structures for program analysis, especially when loops contain multiple paths with complex interleaving executions among these paths. In this paper, we first propose a classification of multi-path loops to understand the complexity of the loop execution, which is based on the variable updates on the loop conditions and the execution order of the loop paths. Secondly, 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 on the variables. The key contribution is to use a path dependency automaton (PDA) to capture the execution dependency between the paths. A DFS-based algorithm is proposed to traverse the PDA to summarize the effect for all feasible executions in the loop. The experimental results show that Proteus is effective in three applications: Proteus can 1) compute a more precise bound than the existing loop bound analysis techniques; 2) significantly outperform state-of-the-art tools for loop verification; and 3) generate test cases for deep loops within one second, while KLEE and Pex either need much more time or fail.

CCS Concepts

•**Theory of computation** → **Program verification; Program analysis;** •**Software and its engineering** → **Software verification; Automated static analysis;** *Software testing and debugging;*

Keywords

Loop Summarization, Disjunctive Summary

1. INTRODUCTION

Analyzing loops is very important for successful program optimizations, bug findings, and test input generation. However, loop analysis is one of the most challenging tasks in program analysis. It is described as the "Achilles' heel" of program verification [12] and a key bottleneck for scaling symbolic execution [47, 54].

*Xiaohong Li is the corresponding author, School of Computer Science and Technology, Tianjin University.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950340

Generally, there are three kinds of techniques for analysing loops, namely *loop unwinding*, *loop invariant inference* and *loop summarization*. The most intuitive method is loop unwinding, where we unroll the loop with a fixed number of iterations (a.k.a. the bound). This technique is unsound and cannot reason about the program behaviors beyond the loop bound. Loop invariant is a property that holds before and after each loop iteration. It is mostly used to verify the correctness of a loop. The limitation is that typically only strong invariants are useful to prove the property; while commonly-used fixpoint-based invariant inferencing [10] is iterative and sometimes time-consuming. It may fail to generate strong invariants, especially for complex loops. In addition, loop invariants often cannot sufficiently describe the effect after the loop and hence are limited to check the property after the loop.

Compared with loop invariants, loop summarization provides a more accurate and complete comprehension for loops [47, 20, 51, 55, 12]. It summarizes the relationship between the inputs and outputs of a loop as a set of symbolic constraints. We therefore can replace the loop fragments with such "symbolic transformers" during program analysis. This leads to a wider range of applications. For example, we can use loop summarization to verify program properties after a loop; and we can use it to better direct test input generation in symbolic execution. Detailed discussion about invariant and summarization can be found in Section 6.

The loop summarization techniques [20, 47] mainly handle *single-path loops* (the simplest type of loops where no branches are present). The recent advances of loop analysis [51, 55] are to perform loop summarization for *multi-path* loops (the loops that contain branches). However, the techniques cannot summarize the interleaving effect among the multiple paths in a loop. The goal of this paper is to reason about the interleaving of multiple paths in the loop and generate a *disjunctive loop summary* (DLS) for such multi-path loops.

As an example, the `while` loop in Fig. 1(a) contains an `if` branch, which makes it a multi-path loop. In addition, the computation in the `if` and `else` branches can impact the outcome of the `if` condition, leading to interleaving of the two paths in the loop. It is the initial values of the variables x, z and n that determine the different possibilities of interleaving between the `if` and `else` branches. For some of the multi-path loops, we can determine what types of interleaving potentially exist and what are the loop summaries for the determined types. Consider Fig. 1(a), let x, z and x', z' be the values before and after loop execution respectively. When the initial values satisfy $x \geq n$, the loop effect is $x' = x \wedge z' = z$. When $x < n \leq z$, the loop effect is $x' = n \wedge z' = z$. When the loop starts with $x < n \wedge z < n$, the loop effect is $x' = z' = n$. Hence, a precise summary of the loop effect should be a *disjunction* that includes all possible loop executions due to different initial values of x, z and n. Thus, the dis-

junctive loop summary 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 summarization in [20, 47, 51, 55], DLS computes the effect of each possible pattern of the loop execution, and it is more specific and fine-grained.

This paper accomplishes two tasks to advance the state-of-the-art loop analysis. First, we proposed a classification for multi-path, single loops (non-nested loops) based on a deep analysis on challenging loops found in real-world software. The 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 handle. The classification is based on two aspects: 1) the update patterns of variables that direct the path conditions and 2) the interleaving patterns among the paths in the loop. Second, we developed a loop analysis framework, named Proteus[1], to summarize the effect of the loops for each type. Proteus takes a loop code fragment and a set of variables of interest as inputs to compute the DLS. The DLS represents a disjunction of a set of path-sensitive loop effects on the variables of interest.

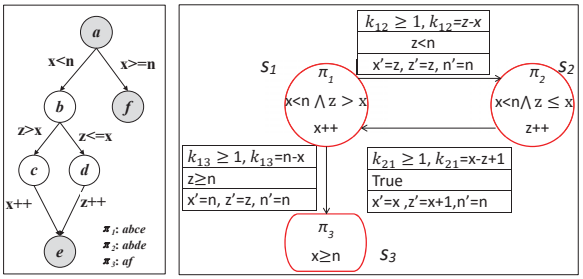
Basically, Proteus generates a fine-grained loop summary in three steps. The first step applies a program slicing on the loop according to the variables of interest so that irrelevant statements are removed to reduce irrelevant paths in the loop. In the first step, we also construct the *loop flowgraph* based on the control flow graph (CFG) of the sliced loop. The second step is a novel technique where we construct *a path dependency automaton (PDA)* from the flowgraph to model the path interleaving. Each state in the PDA corresponds to a path in the flowgraph; and transitions of the PDA capture the execution dependency of the paths. The last step performs a depth-first traversal on the PDA to summarize the effect of each feasible trace in the PDA (which corresponds to an execution in the original loop). The final result is a disjunction of the summaries for all feasible traces. For the challenging loop types that cannot be directly handled, we transform them to the simpler ones with approximation techniques before summarizing them.

We have implemented Proteus and experimentally evaluated the usefulness of summary by applying it to loop bound analysis, program verification and test case generation. We collected 9,862 single loops in total from several open-source projects to understand the distributions of loop types and the complexity of loops in real-world programs. We computed the loop bound for the loops in these projects. The result shows that Proteus can compute a more precise loop bound than the previous techniques [38, 28, 27, 26]. We performed program verification using the benchmark SV-COMP 16 [1]. The result indicates that our approach can summarize 81 (65.85%) of the total 123 programs; among these summarized programs, Proteus can help correctly verify 74 (91.36%) of the programs. Compared to Proteus, SMACK+Corral [31] that achieved the highest correct rate in SV-COMP 16, can only correctly verify 68 (83.95%) of loops. In addition, Proteus only took 75 seconds while SMACK+Corral took more than 7 hours. We also evaluated test case generation by comparing the two configurations of the symbolic execution tools KLEE [8] and Pex [52] with Proteus. Our result shows that with Proteus, it only took less than one second to generate the test cases for all the loops, while KLEE either times out or needs much more time and Pex often throws an exception.

To the best of our knowledge, this is the first work to compute DLS for multi-path loops. The main contributions of this paper are:

1. we propose a classification for multi-path loops of four types to understand the complexity of the loop execution;
2. we propose a path dependency automaton to capture the execution dependency and effects of the paths in multi-path loops;
3. we propose an algorithm to compute DLS based on the path dependency automaton; and
4. we conduct an experimental study to classify the loops in real-world projects as well as to evaluate the usefulness of DLS in three important software applications.

[1]A Greek god who can foretell the future.



(a) Flowgraph (b) Path Dependency Automaton (PDA)

Figure 2: The flowgraph and PDA for the loop in Figure 1(a)

2. OVERVIEW

In this section, we first formally introduce the concepts of flowgraph and loop paths, which are needed to understand the rest of the paper. We then present a classification of multi-path loops we identified and also provide an overview of Proteus that computes the DLS for the identified types. Note that the loops in the scope are multi-path single loops that do not contain other loops inside.

2.1 Preliminaries

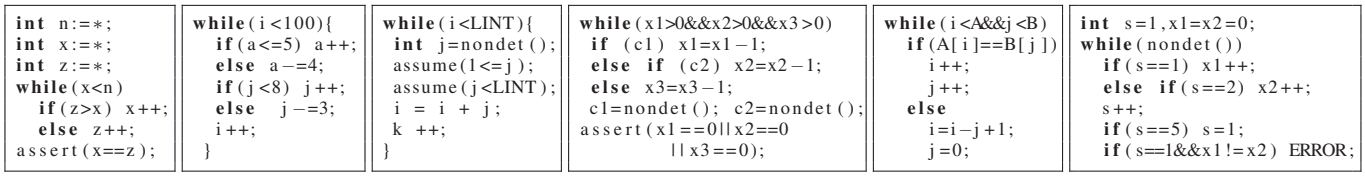
Definition 1. Given a loop, the flowgraph of the loop is a tuple $G = (V, E, v_s, v_t, v_e, \iota)$, where V is a set of vertices, $E : V \times V$ is a set of edges that connect the vertices, $v_s, v_t \in V$ are two virtual nodes that capture the start and end points of each loop iteration, v_e is a virtual node that represents the exit of the loop, and ι is a function assigning every edge $e \in E$ an instruction $\iota(e)$.

A node is a branch node if its out-degree is 2, and the instruction on the edge that starts from the branch node is a boolean condition. For example, Fig. 2(a) shows the flowgraph of the loop in Fig. 1(a). Nodes a and e specify the start and end of the loop iteration respectively, and node f is the exit node. Node b is a branch node, the instructions on the edges (b, c) and (b, d) are conditions.

Intuitively, given a loop flowgraph G, each iteration in the loop from node v_s to v_t is an execution path. We introduce the concept of the *loop path* to better discuss how we model the effect of each loop iteration to compute the summaries of the loop.

Definition 2. Given a loop flowgraph $G = (V, E, v_s, v_t, v_e, \iota)$, a loop path (we call it path here after for simplicity) π in G is a finite sequence of nodes $\langle v_1 v_2 \ldots v_k \rangle$, where $k \geq 1$, $(v_i, v_{i+1}) \in E$, $1 \leq i < k$, $v_1 = v_s$, and $v_k \in \{v_t, v_e\}$. A path is called an *iterative path* if $v_k = v_t$, or an *exit path* if $v_k = v_e$. We use θ_π to denote the path condition of path π, which is a conjunction of the branch conditions along the edges of path π. Given a loop flowgraph G, we use Π_G to denote the set of all paths in G.

In Fig. 2(a), the flowgraph has two iterative paths $\pi_1 = \langle abce \rangle$ and $\pi_2 = \langle abde \rangle$, and one exit path $\pi_3 = \langle af \rangle$. For path π_1, the path condition θ_{π_1} is $x < n \wedge z > x$.



```
int n:=*;
int x:=*;
int z:=*;
while(x<n)
  if(z>x) x++;
  else z++;
assert(x==z);
```

(a) [28]

```
while(i<100){
  if(a<=5) a++;
  else a-=4;
  if(j<8) j++;
  else j-=3;
  i++;
 }
```

(b)

```
while(i<LINT){
 int j=nondet();
 assume(1<=j);
 assume(j<LINT);
 i = i + j;
 k ++;
}
```

(c) [1]

```
while(x1>0&&x2>0&&x3>0)
 if (c1) x1=x1-1;
 else if (c2) x2=x2-1;
 else x3=x3-1;
 c1=nondet(); c2=nondet();
assert(x1==0||x2==0
          ||x3==0);
```

(d) [1]

```
while(i<A&&j<B)
  if(A[i]==B[j])
    i++;
    j++;
  else
    i=i-j+1;
    j=0;
```

(e) [1]

```
int s=1,x1=x2=0;
while(nondet())
  if(s==1) x1++;
  else if(s==2) x2++;
  s++;
  if(s==5) s=1;
  if(s==1&&x1!=x2) ERROR;
```

(f) [5]

Figure 1: Motivating loop programs from the recent work [28, 5] and the SV-COMP 16 benchmark [1]

2.2 Loop Classification

To summarize a multi-path loop, we need three critical pieces of information: (1) value changes in each iteration of one path, (2) the number of iterations of each path, and (3) the execution order of the paths. Since a loop path consists of a finite sequence of nodes, we can perform symbolic analysis to derive value changes at the end of the path. The number of iterations of each path depends on the path condition. If the variables in path condition are induction variables, we usually can reason about the number of iterations. The execution order of loop paths depends on the input of a loop. We found there are patterns that we can summarize to describe the path interleaving.

Based on the above analysis, the difficulties of summarizing a multi-path loop are determined by 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, which we defined the three types *sequential*, *periodic* and *irregular*. In the following, we provide a detailed explanation for the two patterns, and also present the classification of a multi-path loop based on the two. The classification represents *how difficult a multi-path loop can be summarized*.

Patterns of Value Changes in Path Conditions. Given variable x and path π, we write $\Delta^i_\pi x$ to denote the value change of x between the $(i-1)^{th}$ and i^{th} iterations of π. We define the induction variable as follows according to their value change during the execution.

Definition 3. Given a loop flowgraph G, a variable x is an *induction variable* if $\forall \pi \in \Pi_G$, for any i^{th} and j^{th} iterations of π, $\Delta^i_\pi x = \Delta^j_\pi x$. Otherwise, x is a *non-induction variable*.

For an induction variable x, the value change of x is constant in each iteration of π, and we write it as $\Delta_\pi x$. For example, in Fig. 1(a), x is an induction variable as the change of x over each iteration of the loop path π_1, π_2 or π_3 is constant; and we have $\Delta_{\pi_1} x = 1$, $\Delta_{\pi_2} x = 0$ and $\Delta_{\pi_3} x = 0$. Similarly, z and n are also induction variables. Note that it is undecidable to determine induction variables. In our implementation, we perform a conservative static analysis and report one variable as induction variable only when we statically identify that the symbolic change of the variable is constant in each iteration of a path.

Each condition in a path can be transformed to the form of $E \sim 0$, where E is an expression and $\sim \in \{<, \leq, >, \geq, =\}$. For the operator $\neq$, we transform it to $E > 0 \vee E < 0$. We use E as a variable for the ease of presentation. We classify each condition into two types:

- *IV Condition*. A condition is an IV condition if E is an induction variable. For example, the condition $x < n$ in Fig. 1(a) is an IV condition since $\Delta_{\pi_1} E = 1$ and $\Delta_{\pi_2} E = \Delta_{\pi_3} E = 0$, where $E = x - n$.
- *NIV Condition*. A condition is a NIV condition if E is a non-induction variable. For example, the condition $i < A$ in Fig. 1(e) is a NIV condition since $v = i - A$ is non-induction variable.

In some NIV conditions, the value change of E is solely dependent on the input or context before the loop, but not the statements in the loop. For example, the condition in the loop that traverses a data structure is often dependent on the content of the data structure; and a non-deterministic function as a condition cannot determine the value change patterns over the iterations. We call such NIV conditions *input-dependent NIV conditions*. In Fig. 1(e), the condition $A[i] == B[j]$ is an input-dependent NIV condition since the value change of $A[i] - B[j]$ depends on the element contents of A and B. The conditions $c1$ and $c2$ in Fig. 1(d) are also input-dependent NIV conditions as they depend on the non-deterministic function *nondet*.

Table 1: A Classification of Single Loops

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

Patterns of Path Interleaving. We use the concept of *loop execution* to define the patterns of path interleaving in the loop. The *precondition* of the loop specifies the conditions of the variables before entering the loop. Note that, with different preconditions, the loop may have different executions.

Definition 4. Given a loop flowgraph G with a precondition, a loop execution ρ_G is a sequence of paths $\langle \pi_1, \pi_2, \ldots, \pi_i, \ldots \rangle$, where $\pi_i \in \Pi_G$ for all $i \geq 1$. We use $\pi_i \rightarrow^* \pi_j$ to represent a subsequence from π_i to π_j in ρ_G.

If $\exists\, i \neq j$, $\pi_i \rightarrow^* \pi_j \rightarrow^* \pi_i$ is a subsequence of ρ_G, then ρ_G contains a cycle. The cycle is *periodic* if its execution has the pattern $\langle \pi_i^{k_i}, \ldots, \pi_j^{k_j}, \ldots \rangle^+$ (its period is $\langle \pi_i^{k_i}, \ldots, \pi_j^{k_j}, \ldots \rangle$), where k_i and k_j are constant values that represent the execution times of π_i and π_j respectively. For example, in Fig 2(a), the flowgraph contains cycle $\pi_1 \rightarrow^* \pi_2 \rightarrow^* \pi_1$, whose execution is $\langle \langle \pi_1^1, \pi_2^1 \rangle, \langle \pi_1^1, \pi_2^1 \rangle, \ldots \rangle$. Thus, this cycle is periodic and its period is $\langle \pi_1^1, \pi_2^1 \rangle$.

Given a loop flowgraph G with a precondition, we classify a loop execution ρ_G into three types:

- *Sequential Execution*. If ρ_G does not contain any cycle, it is a sequential execution.
- *Periodic Execution*. If all cycles in ρ_G are periodic, it is a periodic execution.
- *Irregular Execution*. If a loop execution is neither sequential nor periodic, we call it an irregular execution. In this case, the loop execution contains cycles; however, the path interleaving pattern for the loop execution cannot be statically determined. Therefore, we cannot easily compute the number of iterations for each path.

Loop Classification. In Table 1, we show a loop classification we defined based on the above two patterns. The first row indicates that we classify a multi-path loop based on whether all the conditions in the loop are *IV conditions* (see Type 1 and 2) or there exists a condition that can be the *NIV condition* (see 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

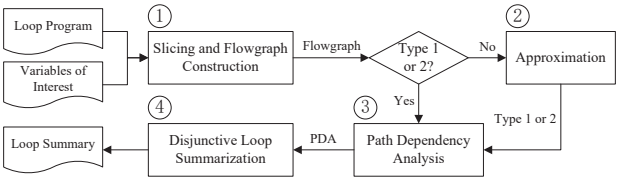


Figure 3: An overview of our framework Proteus

Periodic (see Type 1 and 3) or there exists one loop execution that can be *Irregular* (see Type 2 and 4).

Typically, the loops related to integer arithmetics, e.g., $for(;i<n;i{+}{+})$, often belong to Type 1. The loops that traverse a data structure often belong to Type 3 and 4, as the loop iteration depends on the content of the data structure. Intuitively, loops with NIV conditions such as the ones related to complex data structures tend to have no path interleaving patterns (i.e., irregular execution).

As examples, the loop in Fig. 1(a) is Type 1, as it only contains IV conditions and has a periodic execution. The loop in Fig. 1(b) belongs to Type 2. Although theoretically Type 2 loops should exist, in practice, we did not find any Type 2 loops in real-world programs (details are given in the evaluation section). The loop in Fig. 1(c) contains an NIV condition and its execution is sequential; the loop in Fig. 1(f) contains NIV conditions and has a periodic execution. Thus both of them belong to Type 3. The loops in Figs. 1(d) and 1(e) belong to Type 4 because they contain input dependent NIV conditions that can lead to an irregular execution.

2.3 Loop Analysis Framework

Fig. 3 shows the workflow of Proteus. It takes a *loop program* 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 who 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, 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 can also generate a summary for all the induction variables in the loop.

Proteus mainly consists of four steps to summarize a loop. Step 1 performs a program slicing using the variables of interest as slicing criteria and constructs the flowgraph for the sliced loop program. The loop slicing removes the irrelevant statements, which can help reduce irrelevant paths in the constructed flowgraph and make our summarization more efficient. We implement the loop slicing based on the program dependence graph (PDG) [45] which combines the control flow graph (CFG) and data dependencies and the details can be found in [38]. Then, the flowgraph is constructed based on the CFG by adding the virtual start node, end node and exit node.

From the flowgraph, 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 Type 2; otherwise, the loop belongs to Type 3 or Type 4. Summarizing the non-induction variables is very challenging because of the uncertain value changes, we provide some approximation techniques (Step 2) to transform some of the loops of Type 3 and Type 4 to Type 1 or Type 2. This approximation may cause imprecise summaries but may still be useful and effective in specific applications. We cannot handle Type 3 and Type 4 loops that cannot be approximated.

In Step 3, Proteus extracts the path condition and value changes of the variables from the flowgraph and analyzes the dependency between any two paths. We propose a path dependency automaton (PDA) to capture the execution order and path interleaving patterns of the paths. Note that we can construct the PDA for Type 1 and Type 2 loops. In Step 4, we perform a depth first search on the PDA to check the feasibility of each trace in the PDA and summarize the loop effect if it is feasible. The loop summary is a disjunction of the summaries for all loop executions (i.e., all feasible traces in the loop). The last two steps are our main contributions in this paper, which will be elaborated in Section 3 and 4 respectively.

3. PATH DEPENDENCY ANALYSIS

This section presents the definition of path dependency automaton (PDA) and the algorithm to construct a PDA from a flowgraph.

3.1 Path Dependency Automaton

Definition 5. Given a flowgraph G with a set of induction variables X, the path dependency automaton (PDA) is a 4-tuple $M = (S,\ init,\ accept,\ \hookrightarrow)$, where

- S is a finite set of states, each of which corresponds to a path in Π_G. Each state $s \in S$ is a 3-tuple $(\pi_s,\ \theta_s,\ \Delta_s X)$, where $\pi_s \in \Pi_G$ is the corresponding path, θ_s is the path condition of π_s, and $\Delta_s X$ represents the set of the value changes for all the induction variables after one execution of π_s.
- $init \subseteq S$ is a set of initial states.
- $accept \subseteq S$ is a set of accepting states, which have no successors. An accepting state is called an *exit state* if it corresponds to an exit path, and a *terminal state* if it corresponds to an iterative path.
- $\hookrightarrow \subseteq S \times S$ is a finite set of transitions. We use $s_i \hookrightarrow s_j$ to represent the transition $(s_i, s_j) \in \hookrightarrow$, and introduce a variable k_{ij} (≥ 1) as the *state counter* to indicate that s_i can transit to s_j after k_{ij} executions of s_i. Each transition $s_i \hookrightarrow s_j$ is annotated with a 3-tuple transition predicates $(\phi_{ij},\ \varphi_{ij},\ U_{ij})$. ϕ_{ij} is a constraint about k_{ij}. φ_{ij} is the guard condition, satisfying which, $s_i \hookrightarrow s_j$ will be triggered. Note that ϕ_{ij} and φ_{ij} are conditions on the variables before entering into s_i. U_{ij} is a function computing X', which are the values of variables X after k_{ij} executions of s_i. That is, $X' = U_{ij}(X, k_{ij})$.

Note that a terminal state indicates that, once the execution enters a terminal state, it never transits to other paths, i.e., the loop will execute infinitely on the path. For example, there are two states corresponding to the two paths of the loop $while(i<10)\ i=0$. The state corresponding to the iterative path $i<10 \wedge i=0$ is a terminal state, leading to an infinite execution of the loop.

For a PDA M, a trace τ in M is a sequence of transitions $s_1 \hookrightarrow \ldots \hookrightarrow s_i$, where $s_1 \in init$ and $s_i \in accept$. A trace τ represents a possible loop execution. Note that not all traces in PDA are feasible, and the existence of transitions $s_i \hookrightarrow s_k$ and $s_k \hookrightarrow s_j$ cannot guarantee $s_i \hookrightarrow s_k \hookrightarrow s_j$ is feasible. We use E_M to denote the set of all feasible traces in M, and X_τ to represent the symbolic values of the variables X after the execution of the trace τ.

Example. Fig. 2(b) shows the PDA of the loop in Fig. 1(a). Its flowgraph is given in Fig. 2(a). In the PDA, $S = \{s_1, s_2, s_3\}$ corresponds to the three paths of the loop, $init = \{s_1, s_2, s_3\}$ represents the initial states (marked as red), and $accept = \{s_3\}$ corresponds to the exit path in the loop. For state s_1, it corresponds to path π_1, whose path condition θ_{s_1} is $x<n \wedge z>x$. The value changes along π_1 are $\Delta_{s_1}\{x,z,n\} = \{1,0,0\}$ (here we omit the variables that do not change over the loop iterations). The table above the transition represents the transition predicates. For the transition $s_1 \hookrightarrow s_2$, it indicates that after executing s_1 for k_{12} number of times, the loop

Algorithm 1: ConstructPDA

input : G: flowgraph, *precond*: the precondition of the loop
output: M: PDA

1 Construct states S from paths $\prod_G$;
2 Let X be the induction variables in the flowgraph G;
3 **foreach** $s \in S$ **do**
4 **if** *solve(precond* $\wedge \theta_s$*)= SAT* **then**
5 init := init $\bigcup\{s\}$;
6 **foreach** $s_i, s_j \in S \wedge i \neq j$ **do**
7 Let $k_{ij} \geq 1$ be the state counter for $s_i \hookrightarrow s_j$;
8 Let $X'_{k_{ij}-1}$ be the variables after k_{ij}-1 executions of state s_i;
9 Let $X'_{k_{ij}}$ be the variables after k_{ij} executions of state s_i;
10 $cond := \theta_{s_i}[X'_{k_{ij}-1}/X] \wedge \theta_{s_j}[X'_{k_{ij}}/X]$;
11 **if** *solve(cond)= SAT* **then**
12 $(\phi_{ij}) \leftarrow$ simplify $(cond)$;
13 $(\varphi_{ij}) \leftarrow$ eliminate $(cond, \phi_{ij})$;
14 $\hookrightarrow := \hookrightarrow \bigcup\{(s_i, s_j)\}$;
15 Annotate $(\phi_{ij}, \varphi_{ij}, U_{ij}(X, k_{ij}))$ with the transition $s_i \hookrightarrow s_j$;
16 *accept* is the set of states which have no successors;
17 **return** $M = (S, init, accept, \hookrightarrow)$;

Algorithm 2: SummarizeType1Loop

input : M: PDA, *precond*: precondition
output: S_M: loop summary of M on precondition *precond*

1 Let X be the induction variables in M and *rec* be a summary map;
2 **foreach** $s_i \in init$ **do**
3 $SummarizeTrace(s_i,\ precond \wedge \theta_{s_i},\ X,\ rec)$;
4 **return** S_M

Algorithm 3: SummarizeTrace

input : s_i: current state, tc: current trace condition
X': updated variables, *rec*: a map

1 **if** $s_i \in rec$ **then**
2 Summarize cycle by checking the period;
3 **else if** $s_i \in accept$ **then**
4 **if** s_i *is exit state* **then**
5 $S_M = S_M \bigcup \{(tc,\ \Delta_{s_i} X')\}$.
6 **else**
7 $S_M = S_M \bigcup \{(tc,\ \Delta^{\infty}_{s_i} X')\}$;
8 **else**
9 **foreach** $s_j \in \{s_m \mid s_i \hookrightarrow s_m \in \hookrightarrow\}$ **do**
10 Let $(\phi_{ij},\ \varphi_{ij},\ U_{ij})$ be the predicates on transition $s_i \hookrightarrow s_j$;
11 **if** *solve*$(tc \wedge \varphi_{ij}[X'/X])$*=SAT* **then**
12 $nrec := clone(rec)$;
13 $nrec[s_i] := \{tc \wedge \varphi_{ij}[X'/X],\ U_{ij}(X',\ k_{ij})\}$;
14 $SummarizeTrace(s_j,\ tc \wedge \varphi_{ij}[X'/X],\ U_{ij}(X', k_{ij}),\ nrec)$;

leads to s_2. The first row in the table specifies the constraint for k_{12}: $k_{12} \geq 1 \wedge k_{12} = z - x$ (i.e., ϕ_{12}). The second row is the guard condition φ_{12}, $z < n$, indicating that when the condition $z < n$ is satisfied, s_1 is transited to s_2 after k_{12} number of iterations. The function U_{12} (the third row) is $\{x', z', n'\} = \{z, z, n\}$, suggesting that after executing s_1 for k_{12} times, the symbolic values of x, z and n become z, z and n. Note that the variables x, n, z on each table are not the initial values before the loop, but the values before executing the source state of the corresponding transition.

3.2 Construction of PDA

Algorithm 1 presents the procedure to construct a PDA from the flowgraph. The input of the algorithm includes the loop flowgraph G and the precondition of the loop *precond*. The output is the constructed PDA. At Line 1, we first construct the states S of PDA from Π_G. For each state $s \in S$, we solve the constraint $precond \wedge \theta_s$ by using the SMT solver Z3 [14]. If the result is SAT (i.e., the constraint is satisfied), the state s is added into *init* (Line 4–5).

Then we compute the transition between any two states s_i, s_j in the PDA (Line 6–6). First, we introduce the state counter $k_{ij} \geq 1$ for the transition $s_i \hookrightarrow s_j$, and we also specify the value of variables after $k_{ij} - 1$ and k_{ij} times of execution of s_i as $X'_{k_{ij}-1}$ and $X'_{k_{ij}}$ (Line 7–9). The key observation here is that if a transition between s_i to s_j is feasible, θ_{s_i} should be satisfied with variables $X'_{k_{ij}-1}$ and θ_{s_j} should be satisfied with variables $X'_{k_{ij}}$. We use $\theta[X'/X]$ to represent X are substituted with X' in the condition θ. At Line 10, we get the guard condition for $s_i \hookrightarrow s_j$ by the conjunction of the substituted θ_{s_i} and θ_{s_j}. This computation is effective for Type 1 and Type 2 loops that only contain IV conditions (discussed in Section 4.1). In this case, the change of the variables is linear along the path, i.e., $X'_{k_{ij}-1}$and $X'_{k_{ij}}$ have a linear relation with X.

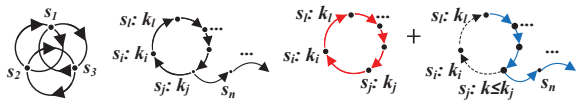
Finally, we use the SMT solver Z3 to solve the generated guard condition at Line 11. If the result is SAT, the transition $s_i \hookrightarrow s_j$ is feasible (UNSAT means there is no transition from s_i to s_j). We simplify the linear inequalities in the guard condition based on an extended Z3 tactics [14] and compute the predicate for k_{ij} (Line 12). At Line 13, we aim to eliminate the variable k_{ij} in *cond* if possible and simplify the guard condition φ_{ij} to use only loop variables. In particular, if θ_{s_i} implies *cond*, φ_{ij} can be simplified to true, which means once entering into s_i, s_i can always transit to s_j. Note that if it cannot be eliminated, it just keeps the original *cond* and does not affect our approach. We add the transition into the set $\hookrightarrow$ (Line 14), and update the variables in X using the state counter k_{ij}. The result $U_{ij}(X,\ k_{ij})$ is used with ϕ_{ij}, φ_{ij} to annotate the transition (Line 15).

Example. Using the previous example in Fig. 2(b), we explain how the transitions are computed. Consider the transition $s_1 \hookrightarrow s_2$ in Fig. 2(b). The table on top of $s_1 \hookrightarrow s_2$ in Fig. 2(b) shows the 3-tuple transition predicates. Let k_{12} be the state counter. The updates of the variables after $k_{12} - 1$ and k_{12} times of executions of path π_1 are $X'_{k_{12}-1}$: $\{x' = x + k_{12} - 1, n' = n, z' = z\}$ and $X'_{k_{12}}$: $\{x' = x + k_{12}, n' = n, z' = z\}$ respectively. We then can compute the conjunction of the two path conditions θ_{s_1} and θ_{s_2} by substituting the variables X with $X'_{k_{12}}$ and $X'_{k_{12}-1}$ respectively, and obtain *cond* as $(x + k_{12} - 1 < n) \wedge (z > x + k_{12} - 1) \wedge (x + k_{12} < n) \wedge (z \leq x + k_{12})$. After the simplification of the inequalities, we can get ϕ_{ij} as $z - x \leq k_{12} < z - x + 1$, i.e., $k_{12} = z - x$. Using this information, we can further simplify *cond* to be $z < n$ (i.e., φ_{12}). The update function U_{12} is $\{x', n', z'\} = \{x + 1 \times k_{12}, n + 0 \times k_{12}, z + 0 \times k_{12}\} = \{z, n, z\}$. Similarly, we can also compute $\varphi_{21} = \ x < n$ for transition $s_2 \hookrightarrow s_1$. However, φ_{21} can be simplified as true since θ_{s_2} implies $x < n$,

4. DISJUNCTIVE LOOP SUMMARIZATION

In this section, we elaborate the algorithm for computing DLS, which is formally defined below.

Definition 6. Given a PDA M generated from a loop flowgraph G and a set of induction variables X, the summary of a trace $\tau \in M$ is denoted as a tuple $(tc_\tau,\ X_\tau)$, where tc_τ is the condition needed to meet when trace τ is feasible, and X_τ is the value of the variables after executing the trace. The loop summary of M, denoted as S_M, is $\bigcup_{\tau \in E_M}\{(tc_\tau, X_\tau)\}$, i.e., the union of all trace summaries in the loop.



(a) Circles (b) Cycle (c) Cycle Execution

Figure 4: Examples of cycle

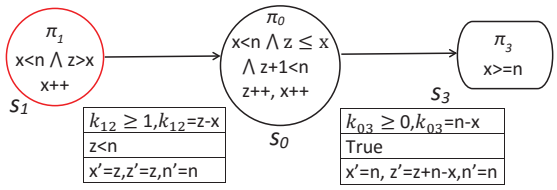


Figure 5: The substitution of a cycle

4.1 Summarization for Type 1 Loops

Algorithm 2 shows the detailed procedure for summarizing Type 1 loops. It takes as inputs the PDA of a loop M, the precondition before the loop *precond*, and returns the loop summary S_M. Let X be the set of the induction variables, which contain the variables of interest, and *rec* be a map to record the summary for the current trace (i.e., from the initial state to the current state) (Line 1). Algorithm 2 traverses each initial state to summarize the feasible traces starting from it by calling Algorithm 3 (Line 3).

Algorithm 3 performs a depth-first search (DFS) on the PDA to summarize each transition of the trace until it reaches an accepting state. Its inputs are the current state s_i, the current trace condition tc for the prefix trace during DFS, the values of variables X' after the previous transition summarization, and the summary map *rec*. Specifically, if the current state s_i is contained in *rec*, a cycle is found and we summarized the cycle by its period (which will be introduced later) (Line 1–2). If s_i is an accepting state, the summarization for one trace is finished (Line 3–7). In particular, if the accepting state is an exit state, tc is the satisfied condition of the trace; the variables X' are updated to $\Delta_{s_i}X'$ in the exit path (Line 5). If s_i is a terminal state, the trace corresponds to an infinite execution. Thus the current variables X' are updated to $\Delta^{\infty}_{s_i}X'$, where ∞ means the infinite update of $\Delta_{s_i}X'$. In the implementation, we use a symbolic infinite value to represent this infinite update.

On the other hand, if s_i is not an accepting state, the algorithm continues to summarize the transitions from s_i to its successors (Line 9–14). For each successor s_j, let $(\phi_{ij},\ \varphi_{ij},\ U_{ij})$ be the transition predicates (Line 10). The guard condition φ_{ij} is updated by substituting its variables X with X'. The constraint $tc \wedge \varphi_{ij}[X'/X]$ is solved by the SMT solver to check whether the current trace can transit to s_j (Line 11). If feasible, the algorithm clones a new map *nrec* for the new branch (Line 12), updates the current trace condition to $tc \wedge \varphi_{ij}[X'/X]$, updates the variables to $U_{ij}(X', k_{ij})$, and stores the current trace summary into *nrec* (Line 13). Then it continues the summarization from state s_j (Line 14).

Example. For the PDA in Fig. 2(b), the precondition is $true$. Starting with the initial state s_3, Algorithm 3 reaches an exit state. Thus, the trace condition is $x \ge n$, the variables do not change, and the summary for the trace s_3 is $(x \ge n,\ x' = x \wedge z' = z \wedge n' = n)$. Starting with the initial state s_1 which has two successors, the initial trace condition is $x < n \wedge z > x$. Consider the transition $s_1 \hookrightarrow s_3$, the execution reaches an exit state after this transition. The trace condition is updated to $x < n \wedge z > x \wedge z \ge n$, simplified as $x < n \le z$. The variables are updated to $\{x' = n \wedge z' = z \wedge n' = n\}$. Thus, the summary for trace $s_1 \hookrightarrow s_3$ is $(x < n \le z,\ k_{13} = n - x \wedge x' = n \wedge z' = z \wedge n' = n)$.

Cycle Summarization. Summarizing a cycle is challenging since the execution number of each state is uncertain during the executions of the cycle. Multiple connected cycles are challenging due to the interleaving of the cycles. For example, Fig. 4(a) shows the interleaving of three cycles, which represents the dependencies among the three paths in Fig. 1(d). The execution order of such connected cycles are often undecidable. Hence, the loops that contain multiple connected cycles are regarded as irregular execution. In our experiments, we found the cycles that connected are all aperiodic, and there is no loop containing connected periodic cycles.

Table 2: The summarization process for $s_1 \hookrightarrow s_2 \hookrightarrow s_1 \hookrightarrow s_2$

trace	*tc*	X'
$\boldsymbol{s_1} \hookrightarrow s_2$	$x < n \wedge$ $z > x \wedge z < n$	$k_{12} = z\text{-}x$, $x' = z,\ z' = z,\ n' = n$
$s_1 \hookrightarrow \boldsymbol{s_2} \hookrightarrow s_1$	$x < n \wedge z > x$ $\wedge z < n \wedge true$	$k_{12} = z\text{-}x,\ k_{21} = 1,$ $x' = z,\ z' = z{+}1,\ n' = n$
$s_1 \hookrightarrow s_2 \hookrightarrow \boldsymbol{s_1} \hookrightarrow s_2$	$x < n \wedge z > x$ $\wedge z < n \wedge z{+}1 < n$	$k_{12} = z - x = 1,\ k_{21} = 1,$ $x' = z{+}1,\ z' = z{+}1,\ n' = n$

However, if the cycle is periodic, each execution of the cycle has the same pattern, i.e., the period (see Section 2). Then we can compute the effect of each period and abstract the cycle as a new state since the execution of the cycle is periodic. Each execution of the new state represents a full execution of the cycle (i.e., the period), and the state counter of the new state represents the execution number of the cycle. For example, Fig. 4(b) is a periodic cycle, its period is $\langle s_l^{k_l}, \ldots, s_j^{k_j}, s_i^{k_i} \rangle$, and it has one successor s_n. Fig. 4(c) shows the specific execution pattern of Fig. 4(b). The execution consists of two parts: 1) k (≥ 0) executions of the complete cycle (the red part), and 2) one execution of the remaining chain (a partial cycle) $s_l \hookrightarrow \ldots \hookrightarrow s_j \hookrightarrow s_n$ (the blue part). The red part is abstracted as a new state, each of whose execution represents $k_l,\ \ldots,\ k_j,\ k_i$ executions of the states $s_l,\ \ldots,\ s_j,\ s_i$ respectively.

Example. Table 2 shows the summarization for $s_1 \hookrightarrow s_2 \hookrightarrow s_1 \hookrightarrow s_2$ in Fig. 2(b). In the third row, since s_1 is already in *rec*, a cycle $c = s_2 \hookrightarrow s_1 \hookrightarrow s_2$ is detected. We can learn from $k_{12} = k_{21} = 1$ that c is periodic and the period is $\langle s_2^1,\ s_1^1 \rangle$. Fig. 5 shows the PDA after substituting the cycle as new state s_0. The path condition θ_{s_0} is $x < n \wedge z \le x \wedge true \wedge z' < n'$ (where $z' = z + 1, n' = n$) and the variables x and z increase by one in each iteration of the cycle. Then we compute the path dependency for $s_0 \hookrightarrow s_3$. Finally, the trace of the loop execution (acyclic) is $s_1 \hookrightarrow (s_2 \hookrightarrow s_1)^+ \hookrightarrow s_3$[2], and its summary is $(x < z < n,\ k_{12} = z - x \wedge k_{03} = n - z \wedge x' = n \wedge z' = n \wedge n' = n)$. Similarly, we can also compute the summary for another trace $s_2 \hookrightarrow (s_1 \hookrightarrow s_2)^* \hookrightarrow s_1 \hookrightarrow s_3$ as $(z \le x < n,\ k_{21} = x - z + 1 \wedge k_{01} = n - x - 1 \wedge k_{13} = 1 \wedge x' = n \wedge n' = n \wedge z' = n)$.

4.2 Summarization for Types 2, 3 and 4 Loops

It is non-trivial to summarize Type 2, 3 and 4 loops precisely as they contain NIV conditions, irregular executions, or both. We introduce several approximation techniques to Proteus to facilitate the summarization, which are still effective for specific applications.

NIV Condition. NIV conditions are difficult to be summarized because of the unpredictable value change for non-induction variables. We use different strategies for three kinds of NIV conditions.

1) If the variables in a condition are monotonically increased (or decreased) and we only care about the path dependencies (e.g., in loop bound analysis and termination analysis), we approximate the

[2]This trace is a simplification of $s_1 \hookrightarrow (s_2 \hookrightarrow s_1)^* \hookrightarrow s_2 \hookrightarrow s_1 \hookrightarrow s_3$ as the chain after the cycle is also one complete cyclic execution.

change of the value as increased by one (or decreased by one). This approximation makes the variable becomes an IV. The summary can still be used to get a safe result for some applications.

Example. The variable i in the loop in Fig. 1(c) is a non-induction variable but it is always increased. If we want to compute a bound for the loop, we approximate that i is increased by one in each iteration; and we compute the loop bound as $LINT$, which is a safe bound.

2) Some NIV conditions are from the data structure traversal (e.g., list), which are difficult to summarize. In bound analysis, we perform a pattern-based approach to capture some of them; and the string loops can also be summarized with our technique in [55].

Example. We use the attribute $size(li)$ to describe the loop iterations for the data structure traversal loop like $for(;li! = null;li = li \rightarrow next)$, then the loop can be converted to $for(;li < size(li);li++)$. Similarly, $length(str)$ can be used to describe for the string traversal loop like $for(char * p = str; p! = \text{`}\backslash 0\text{'}, p{+}{+})$.

3) For input-dependent NIV conditions, they can always be satisfied in any iteration since their values are dependent on the input or context but not the loop execution. Thus, we abstract them as $true$.

Example. The loop $assume(i > 0);\ while(i >= 0 \&\& v[i] > key)$ $i--$; contains one iterative path and two exit paths. The condition $v[i] > key$ is an input-dependent NIV condition. After abstracting it as true, we get π_1: $\{i >= 0 \wedge true,\ i--\}$, π_2: $\{i >= 0 \wedge true\}$ and π_3: $\{i < 0\}$. Then we can summarize it as: (1) the trace summary for $\pi_1 \hookrightarrow \pi_2$ is $(i > 0,\ 1 \leq k_{12} \leq i \wedge i' = i - k_{12})$ and (2) the trace summary for $\pi_1 \hookrightarrow \pi_3$ is $(i > 0,\ k_{13} = i + 1 \wedge i' = -1)$. Note that this loop summary is also precise. Imprecision can be caused when the content of data structures are updated before or in the loop. We will discuss the imprecision in Section 5.

Irregular Execution. For loops with irregular path executions, the interleaving pattern can be arbitrary and thus cannot be determined. Hence, we do not consider the interleaving order between any two paths, but consider the total effect of each path during the whole loop execution by introducing a path counter [55] k_i for each path π_i. Each path counter k_i can be used to compute the values of the induction variables after k_i executions of the loop.

Intuitively, loops with irregular execution satisfies the following condition: assume π_i can transit to the exit path, then $\forall j \neq i$, after k_j iterations of π_j and $k_i - 1$ iterations of π_i, it will satisfy the exit condition of the loop; and after k_j iterations of π_j and k_i iterations of π_i, it will violate the exit condition.

Example. The loop in Fig. 1(d) has irregular executions with three paths. We introduce path counters k_1, k_2 and k_3 to represent their total execution count. The variables after the loop can be $x_1' = x_1 - k_1 \wedge x_2' = x_2 - k_2 \wedge x_3' = x_3 - k_3$. For the path π_1 which can transit to the exit path, *the variables after* $k_1 - 1$ *iterations of* π_1 *satisfy the exit condition, and the variables after the* k_1 *iterations of* π_1 *violate the exit condition*, which implies the constraints $x_1 - (k_1 - 1) > 0 \wedge x_2' > 0 \wedge x_3' > 0 \wedge x_1' \leq 0 \wedge x_2' > 0 \wedge x_3' > 0$. By simplifying the constraints, we can get x_1'=0 $\wedge\ x_2' > 0 \wedge x_3' > 0$. Similarly, if the last iteration is path π_2 or path π_3, we can compute x_2'=0 or x_3'=0. The summary can be used to verify the property after the loop successfully.

4.3 Discussion

Precision. Our summarization for Type 1 loop is precise with respect to the following aspects. First, we perform an equivalent translation from a loop program to its PDA. Second, with the induction variables and state counters, the summarization is an accumulation of the variable updates in the execution of the trace; and there is no approximation involved when producing DLS from PDA in Algorithm 2. Third, DLS is disjunctive and thus fine-grained, we consider all of the possible loop execution patterns under different preconditions of the loop. Types 2-4 loop summarization may introduce imprecision because approximation is used on non-induction variables. However, it can still be useful and precise in certain applications.

Limitation. In this paper, we mainly focus on the systematic summarization for Type 1 loops. Summarization for non-induction variables and nested loops is still an area of open interest. Nested loops are challenging when changes in inner loops make the variables become non-induction variables in outer loops. The problem is then equivalent to summarizing non-induction variables in unnested loops. We will leave the systematic summarization for non-induction variables and nested loops in the future work.

5. EVALUATION

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

We have implemented Proteus using LLVM 3.4 [37] and SMT solver Z3 [14]. We planned the following experiments to achieve our evaluation goals. In the first experiment, we selected five open source projects of different categories, 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 studied the loop classifications for these programs. The results help understand the capabilities of Proteus in solving loops in real-world programs. In the next set of experiments, we applied Proteus for loop bound analysis, program verification, and test input generation. The experimental results are discussed in the following sections.

5.1 Classification of Real-World Loops

In Table 3, the programs under study are listed in the first column. Under *Total (Nest)*, we list a total number of loops discovered, which includes both single loops and nested loops (the number of nested loops is listed in the parenthesis). For the nested loops, we classify only their inner loops. 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. The Type 2 loop shown in Fig. 1(b) is an constructed example. Theoretically Type 2 loops do exist; however, we believe that such loop is difficult to understand and maintain, and the developers typically do not write such code.

The last row of the table summaries the results for all the benchmarks, and the 9862 programs can be classified in less than five minutes. We show that for the five projects under study, we found that 33.87% of the loops belong to Type 1, 19.49% is Type 3 and 46.64% belong to Type 4. Type 4 loops are most common loops, and they are mainly caused by the usage of data structures and arrays in the loop. Type 1 loops are the second most common category. For example, gmp is an arithmetic library and many of its loops are Type 1. Proteus is able to handle such category precisely. With the techniques of demand-driven analysis and approximation, Proteus is also able to handle some of the Type 3 and 4 loops. By further investigating these loops, we found that the dominance of Type 1 and Type 4 loops makes sense—-when a multi-path loop contains NIV conditions, the loop execution is often irregular (Type 4 loops); and when a loop's conditions 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.

In summary, our loop classification can provide a better understanding of real-world loops with respect to the four loops types. It

Table 3: Results of loop classification

Projects	Total (Nest)	Type1	Type3	Type4
coreutils	4977(2620)	1401(28.14%)	812(16.32%)	2764(55.54%)
gmp	1088(114)	920(84.56%)	83(7.63)	85(7.81%)
pcre2	542(373)	188(34.69%)	157(28.97%)	197(36.34%)
libxml	2117 (1150)	640(30.23%)	566(26.74%)	911(43.03%)
httpd	1138 (468)	191(16.78%)	304(26.71%)	643(56.50)
Total	9862 (4725)	3340(33.87%)	1922(19.49%)	4600(46.64%)

Table 4: Results of loop bound analysis

Projects	Type1	Type3	Type4	Total
coreutils	1401(100%)	568(69.96%)	2578(93.27%)	4547(91.36%)
gmp	920(100%)	22(26.51)	57(67.06%)	999 (91.81%)
pcre2	188(100%)	107(68.12%)	136(69.04%)	431 (79.52%)
libxml	640(100%)	384(67.84%)	880(96.71%)	1904 (89.94%)
httpd	191(100%)	219(72.04%)	508(79.00)	918 (80.67%)
Total	3340(100%)	1300(67.64%)	4159(90.41%)	8799 (89.22%)

also guides the design decision in Proteus to primarily focus on Type 1 Loops and to use approximation to handle other loop types.

5.2 Application on Loop Bound Analysis

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 perform summarization on the simplified loop and add up the state counters for each state along feasible traces as a loop bound.

In our experiment, we found that slicing is an effective technique to improve the capability of the loop summarization for a given problem. For example, we found that 83% of the loops in coreutils is simplified as a result of slicing and 69.24% of the paths that are irrelevant to bound analysis are pruned. Based on these sliced loop programs, we can compute the bounds for 8799 (89.22%) loops in the projects. Table 4 provided detailed results for each benchmark. Under *Type1*, *Type3* and *Type4*, we list the percentage of the loops of Type 1, Type 2 and Type 3 where we successfully compute the loop bounds. Under *Total*, we show the percentage of all the loops where we can compute the loop bounds. We found that all these loops are summarized in less than 20 minutes totally.

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

- *NIV Conditions*. The loops has non-induction variables whose value changes are not always increased or decreased in all loop paths, e.g., for conditions that contain function calls. 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 Executions*. 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, and thus the bound is non-deterministic.

We also tried to compare our loop bound analysis results with the current techniques [27, 26, 28]. 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 value change of the variables in some paths is not exactly one, we can compute a more precise bound than them since we summarize the change. For example, in the loop $while(i<n)\ i+=2$ (suppose $i=0$ and $n>0$), the technique in [28] computes the bound as n while we compute the bound as $\lceil n/2 \rceil$. 2) Our approach can compute a fine-grained bound for each possible loop execution trace with the disjunctive summary. 3) Our approach not only computes the bound for the loop, but also computes the bound for each path. This is very crucial and useful in some applications. For example, for worst case execution time (WCET) analysis [53], it is easy to compute the whole execution time based on the path bounds (given the estimated execution time for each path). Differently, the techniques in [27, 28] can compute bounds for some nested loops while we only consider single loops.

```
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)

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

(b)

Figure 6: Loop examples for evaluation

We also found one imprecise loop bound computed by [26] (i.e., Example 3 in Fig.4 in [26]). That loop is shown in Fig. 6(a), which contains interleaving among its multiple paths. Assume that path π_1 takes the if statements (the true branch), π_2 takes the *else* statements (the false branch), and π_3 is the exit path. The loop has only one execution trace $\pi_1 \hookrightarrow (\pi_2 \hookrightarrow \pi_1)^* \hookrightarrow \pi_2 \hookrightarrow \pi_3$, whose summary is $(i=0 \wedge j=0 \wedge 0<m<n,\ k_{12}=m \wedge k_{02}=n-1 \wedge k_{23}=1 \wedge j'=0 \wedge i'=n \wedge m'=m \wedge n'=n)$. Its periodic execution executes π_1 for m times and π_2 for once. Thus, the bound is $m+(m+1)*(n-1)+1=n\times m+n$. However, the result in [26] is $n\times m$.

In summary, using DLS, we can compute a more precise and fine-grained loop bound than the existing loop bound analysis techniques.

5.3 Application on Program Verification

In this experiment, we apply DLS to program verification. We used the benchmark *Loops* in Competition on Software Verification 2016 (SV-COMP 16) [1], which has 5 loop categories. This benchmark contains small but non-trivial loops. Note that the *loop-inv* category 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, which represent the state of the art. CBMC [9] is the basic BMC-based verification tool, and CBMCAcc [36] is the latest work to improve the capability of CBMC on loops with a trace automata. S-MACK+Corral [31], CPAchecker-LPI [3] and SeaHorn [30] are the top tools with respect to correct rate in SV-COMP16 (CPAchecker-LPI achieved the best score in SV-COMP16 for *Loops*). Note that we select the tools based on correct rate rather than the score in SV-COMP16 since we compare the number of correctly verified loop programs. We also select the CPAchekcer based on predicate analysis [5] and CPAchecker-Kinduction based on K-induction [5].

We configured CBMC as in [36], SMACK+Corral, CPAchecker-LPI and SeaHorn as in the competition [1], and CPAchecker as in [5]. All of them were configured with a timeout of 15 minutes. Since CBMCAcc is currently not publicly available, we only used the experimental results from [36] to do the comparison.

Table 5 shows the verification results of those techniques together with the loop summarization statistics. Column *Bench* shows the involved loop categories. Columns *NV*, *AR* and *NL* respectively list the number of loops that cannot be summarized because of non-induction variables, array variables and nested loops. Column *SM* lists the number of loops that can be summarized. Column *TT* reports the total programs (each program contains one loop) in each loop category. Columns *C* report the number of programs that can

Table 5: Verification results of CBMC, CBMC-Acc, CPAchecker, SeaHorn and Proteus

Bench	NV	AR	NL	SM	TT	Proteus		CBMC B = 100		CBMCAcc (100, 3)		CPAchecker Predicate		CPAchecker k-induction		CPAchecker LPI		SMACK +Corral		SeaHorn	
						C	T(s)	C	T(s)	Acc	C	C	T(s)	C	T(s)	C	T(s)	C	T(s)	C	T(s)
loops	4	16	4	40	64	37	35	22	107	22	31	35	2043	33	3050	32	2073	37	5211	31	19
loopacc	5	4	2	24	35	23	19	4	6	24	24	11	9943	12	11743	17	7265	14	12691	12	4601
looplit	2	1	1	12	16	9	14	3	33	-	-	11	1137	5	6510	12	206	12	7052	12	687
loopnew	2	0	1	5	8	5	7	0	2	-	-	0	4506	2	2704	3	1822	5	659	4	903
Total	13	21	8	81	123	74	75	29	148	46	55	57	17629	52	24007	64	11366	68	25613	59	6210

be correctly verified by the techniques, while columns *T* list their time overhead. Column *Acc* gives the number of loops that can be accelerated by the CBMCAcc tool. Here we only compared the programs whose loops can be summarized because our goal is to show the loop summary is complement to these tools to improve their capability on loops.

We compared the verification results with the 81 loops that can be summarized by Proteus. When the bound is set to 100, CBMC can correctly verify 29 (35.80%) loops in 148 seconds. A large number of loops cannot be verified correctly since the bound is not large enough. We also set the bound to 1000, it can correctly verify 35 (43.21%) loops but takes 756 seconds (we omit this from the table for the space limitation). On the other hand, with DLS, Proteus can correctly verify 74 (91.36%) loops within 75 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. The average time for computing DLS for each loop is 0.81 seconds. The results indicate that BMC are often less effective, and our technique can correctly verify more loop programs with less time overhead.

To compare with CBMCAcc, we only show the results for loop categories *loops* and *loop-acc* since they only used these two categories in their paper [36]. In their experiments, the bound was set to 3 if the loop could be accelerated; otherwise, the loop was verified by CBMC with the bound being 100. From their experimental results, we know it can accelerate 22 (55%) of the 40 loops in category *loops* and 31 (77.5%) loops can be verified correctly, while our technique can verify 37 (92.5%) loops. The loops they fail to accelerate are mostly multi-path loops containing complex interleaving. The 24 loops in category *loop-acc* have deep iterations but one single path. Thus, CBMC-Acc can accelerate and verify all of them. Our technique can verify 23 loops, the incorrect one is caused by the imprecision in approximation of input-dependent NIV condition. The results indicate that our technique can handle complex interleaving based on the PDA while CBMCAcc often fails.

Compared with other tools, 57 (70.37%) loops can be correctly verified in 17629 seconds for predicate analysis in CPAchecker, 52 (64.20%) loops in 24007 seconds for *k*-induction, and 64 (79.01%) loops in 11366 seconds for LPI. SeaHorn takes 6210 seconds to correctly verify 59 (72.84%) loops. SMACK+Corral can correctly verify 68 (83.95%) loops in 25613 seconds. Note that the time overhead of CPAchecker, SMACK+Corral and SeaHorn is very large because some programs time out. The results indicate that our technique slightly outperforms these top tools on effectiveness, and significantly outperforms them on performance.

The incorrect verification results of our techniques are caused by the potentially imprecise summaries with approximation. For example, the program in Fig. 6(b), taken from category *loops*, has a input-dependent NIV condition. Our technique approximates the condition $a[j]$!=3 as *true* and thus finds a counterexample *j*==*SIZE*. Actually, the content of array *a* is changed at Line 2, which makes the property *j*<*SIZE* always *true*.

In summary, using DLS, we can correctly verify more programs with less time overhead than existing tools for those loops that we can summarize. Therefore, our loop summary can be en effective complementary to existing tools.

Table 6: Test case generation results of KLEE, Pex and Proteus

Tool	functio ns_false	phases _false	overfl ow_true	multiv ar_false	simple _false1	simple _false2
KLEE	23 min	T/O	T/O	11.97 s	22 min	0.02 s
Pex	F	F	F	F	F	0.11 s
Proteus	0.06 s	0.18 s	0.04 s	0.05 s	0.03 s	0.03 s

5.4 Application on Test Case Generation

In this experiment, we apply DLS to test case generation. We did not compare with other summarization techniques [47, 20, 51] since their tools are not available, and the comparison of the approaches are discussed in the related work. We compared the performance of our technique with the symbolic execution tools KLEE [8] and Pex [52] using the loops in *loop-acc*, which contain deep loops (with large loop iterations). A test case is generated for the assertion after the loop to be true by using KLEE, Pex and our technique. 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 six programs. For the other 18 programs, five of them are the corresponding patched versions of the selected programs and the results are similar; and 13 of them do not have very deep iterations (about 1024 iterations) and the results using the three tools are all less than one second. In the table, T/O represents that KLEE 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.

Among the six programs, the program phases_false has a multi-path loop, and the other five programs contain simple loops, each of them only contains one statement. The results show that even for the simple loops, KLEE timed out for two programs and took much more time for three programs. Pex failed to generate test cases for five programs. This is because symbolic execution consumes much time to keep unfolding the 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 tool by utilizing the summary during symbolic execution. We leave it as our future work to integrate disjunctive loop summarization into symbolic execution.

6. RELATED WORK

Loop invariant is a property hold at the beginning or at the end of each loop iteration (including the exit of the loop). On the other hand, loop summarization focuses on capturing the relations of variables at the entry of the loop and at the exit of the loop, which can also generate symbolic constraints at the exit of the loop. Comparing to loop invariants, loop summaries are more precise and more rich. Hence, computing summary is more challenge than invariant.

In the following, we present the related work on the two areas, and discuss loop analysis in different applications.

6.1 Loop Summarization

Several techniques have been proposed to summarize the loop effect [47, 20, 51, 55, 12]. LESE [47] 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 [20] detects loops and induction variables on the fly and infers the simple partial loop invariants and generates pre- and post-conditions as loop summaries. These techniques only focus on single-path loops.

APC [51] 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 [55] summarizes multi-path string loops using path counters. It extracts the string pattern from each path and then generates the string constraints. Both APC and S-Looper 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 we compute fine-grained DLS which none of the existing techniques have done. In [21], they compute the *may* and *must* summary compositionally and use them for program verification. However, they do not compute the loop summary and loops are handled with invariants.

6.2 Loop Invariant Detection

A number of advances have been made on loop invariant inference [2, 11, 4, 34, 32, 22, 23, 49, 43, 29, 50]. Most of them are based on abstract interpretation [10], 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 [46, 40, 41, 48, 32] 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 [22, 23] are based on octagons and polyhedra, and it cannot compute complete disjunctive invariants. The technique in [49] decomposes a multi-path loop into several single loops, which is difficult to handle complex interleaving. The technique in [43] uses dynamic analysis to generate disjunctive invariants over program trace points, but it is often hard to compute effective trace points for each invariant. The template-based technique [25] needs user-provided templates, and thus is not fully automatic. The techniques in [19, 35] synthesize invariants using templates and learning techniques, which are not sufficient and precise for some properties.

Compared with loop summary, loop invariant has several limitations. First, it cannot guarantee to provide the required invariants (strong and precise enough). Our summarization computes stronger constraints and can also be used to infer invariants [18]. Second, invariant is more suitable for checking properties inside the loop. It cannot describe the loop effect (i.e., postcondition). Proteus can compute symbolic values for variables after a loop and thus can be used in symbolic execution. Finally, most techniques focus on conjunctive invariant, which cannot represent disjunctive properties.

6.3 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. The existing symbolic execution tools [8, 52] mainly handle loops by unrolling the loop, and thus are omitted here.

Loop Bound Analysis. Lokuciejewski et al. [38] compute the loop iteration counts based on abstract interpretation [10]. Their polytope-based approach assumes that the variable in the loop exit condition must increase in each loop iteration and cannot handle the loops in Fig. 1(d) and 1(e). Gulawani et al. [24] compute bounds for multi-path loop based on user annotations. Gulwani et al. [27] 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. [26] use control-flow refinement and progress invariants to estimate loop bounds. Control-flow refinement is similar to PDA, but PDA contains more information and is more specific than control-flow refinement. Its bound computation relies on the standard invariant generator and the result is usually inequalities. Gulwani et al. [28] 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 [39, 9, 13]. It is mainly used to find property violations based on SAT solvers [17, 7, 14], but it can not prove safety properties soundly. Kroening et al. [36] 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 technique in [5] combines 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 [16] supports combined-case k-induction [15] but needs to set k manually. However, split-case k-induction [42, 5] can change k iteratively. ECBMC [42] assigns non-deterministic values to loop termination condition variables, making induction hypothesis too weak and unsound. PKIND [33], CPAchecker [5] and KIKI [6] 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 the loop summary effectively, as shown in our experimental results.

7. CONCLUSIONS

In this paper, we propose a classification for multi-path loops to understand the complexity of loops. Based on the classification, we propose a path dependency automaton to describe the executions of the paths in a loop as well as a loop analysis framework Proteus to perform disjunctive summarization for different types of loops. To the best of our knowledge, this is the first work that can identify different execution patterns of a loop, and compute disjunctive loop summary for multi-path loops with complex path interleaving. In the future, we plan to extend Proteus to support nested loops, summarize Type 3 and 4 loops with abstraction approaches and apply it to more applications such as detection of loop-related performance bugs [44] and analysis of the worst-case execution time [53].

8. ACKNOWLEDGMENTS

This research is supported in part by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30), by the National Science Foundation of China (No. 61572349, 61272106), by the US National Science Foundation (NSF) under Award 1542117 and by DARPA under agreement number FA8750-15-2-0080.

9. REFERENCES

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

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

[3] Lpi. http://lpi.forge.imag.fr/.

[4] M. Antoine. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[5] D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *CAV*, pages 622–640, 2015.

[6] M. Brain, S. Joshi, D. Kroening, and P. Schrammel. Safety verification and refutation by k-invariants and k-induction. In *SAS*, pages 145–161, 2015.

[7] R. Brummayer and A. Biere. Boolector: An e?cient smt solver for bit-vectors and arrays. In *TACAS*, pages 174,177, 2009.

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

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, pages 168–176, 2004.

[10] P. Couso and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[12] K. Daniel, S. Natasha, T. Stefano, T. Aliaksei, and W. C. M. Loop summarization using state and transition invariants. *Form. Methods Syst. Des.*, pages 221–261, 2013.

[13] P. G. de Aledo and P. Sanchez. Framework for embedded system verification(competition contribution). In *TACAS*, pages 429–431, 2015.

[14] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[15] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *SAS*, pages 351–368, 2011.

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

[17] Z. Duan, C. Tian, and L. Zhang. A decision procedure for propositional projection temporal logic with infinite models. In *Acta Informatica*, pages 43–78, 2008.

[18] C. A. Furia and B. Meyer. Inferring loop invariants using postconditions. In *Fields of Logic and Computation*, pages 277–300, 2010.

[19] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.

[20] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33, 2011.

[21] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56, 2010.

[22] D. Gopan and T. Reps. Lookahead widening. In *POPL*, pages 452–466, 2006.

[23] D. Gopan and T. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.

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

[25] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstractinterpretation. In *TACAS*, pages 474–488, 2006.

[26] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[27] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[28] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[29] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, pages 634–640, 2009.

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

[31] A. Haran, M. Carter, M. Emmi, A. Lal, S. Qadeer, and Z. Rakamarica. Smack+corral: A modular verifier. In *TACAS*, pages 451–454, 2015.

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

[33] T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. In *PDMC*, pages 55–62, 2011.

[34] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[35] 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*, pages 328–343, 2010.

[36] D. Kroening, M. Lewis, and G. Weissenbacher. Proving safety with trace automata and bounded model checking. In *FM*, pages 325–341, 2015.

[37] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[38] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract. In *CGO*, 2009.

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

[40] D. Monniaux. Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151, 2009.

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

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

[43] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *ICSE*, pages 608–619, 2014.

[44] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *ICSE*, 2015.

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

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

[47] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA*, pages 225–236, 2009.

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

[49] R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, pages 703–719, 2011.

[50] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, pages 388–411, 2013.

[51] J. Strejček and M. Trtík. Abstracting path conditions. In *ISSTA*, pages 155–165, 2012.

[52] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *NDSS*, pages 134–153, 2008.

[53] 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.

[54] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *ASE*, pages 246 – 256, 2013.

[55] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: Automatic summarization for multipath string loops. In *ISSTA*, pages 188–198, 2015.