Abstracting Path Conditions

Jan Strejček Marek Trtík
Faculty of Informatics, Masaryk University
Brno, Czech Republic
{strejcek,trtik}@fi.muni.cz

ABSTRACT

We present a symbolic-execution-based algorithm that for a given program and a given program location in it produces a nontrivial necessary condition on input values to drive the program execution to the given location. The algorithm is based on computation of loop summaries for loops along acyclic paths leading to the target location. We also propose an application of necessary conditions in contemporary bug-finding and test-generation tools. Experimental results on several small benchmarks show that the presented technique can in some cases significantly improve performance of the tools.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification;
D.2.5 [**Software Engineering**]: Testing and Debugging;
F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Reliability, Verification, Algorithms

Keywords

Symbolic execution, Path conditions, Program location reachability, Tests generation

1. INTRODUCTION

Symbolic execution [19, 18] is enjoying a renaissance during the last decade. The basic idea of the technique is to replace input data of a program by symbols representing arbitrary data. Executed instructions then manipulate expressions over the symbols instead of exact values. Symbolic execution further produces, for each path in a program starting in the initial location, a formula called *path condition*, i.e. a necessary and sufficient condition on input data to drive the execution along the path.

Symbolic execution serves as a basis in many successful tools for test generation and bug finding, for example KLEE [4], EXE [5], PEX [28], SAGE [11], or CUTE [26]. Such tools explore real program paths to cover program locations by tests. There is typically huge (or even infinite) number of real paths, even for very small and simple programs. To cover a given program location by a test then becomes a serious problem. In this paper, we present a dedicated algorithm, which is supposed to help the tools to cover the given program location quickly, and hence improve their performance.

For a given program and a given program location in it, our algorithm computes a nontrivial necessary condition on input values to drive the program execution to the given location. The basic idea of this algorithm is to replace each program loop by a summary of its effect on both, program variables and path conditions. The algorithm is intuitively explained in Section 2 and precisely described in Section 3.

The algorithm usually produces necessary conditions with quantifiers. In spite of recent advances in SMT solving, current SMT solvers often fail to quickly decide satisfiability of quantified formulae. We employ a specific structure of constructed necessary conditions and introduce a transformation of a quantified necessary condition into a more general but quantifier-free necessary condition. The transformation is presented in Section 4.

Section 5 proposes possible applications of necessary conditions in the test-generation tools. In principle, the application of necessary conditions can speed up recognition of unreachable locations as well as discovery of tests reaching uncovered locations. In Section 6 we discuss experimental results, depicting running times of our algorithm and PEX on small set of benchmarks. The purpose of the experiments is to show that PEX could benefit from our algorithm.

Finally, Section 7 discusses some related work and Section 8 concludes the paper.

2. OUTLINE OF THE ALGORITHM

An intuitive explanation of the algorithm is illustrated on the following simple program, where we want to compute a necessary condition to reach the assertion on the last line.

```
k = 0;
for (i = 3; i < n; ++i) {
   if (A[i] == 1)
      ++k;
}
if (k > 12)
   assert(false);
```

The relevant part of the program is represented as the flowgraph of Figure 1, where the node h corresponds to the target location.

Our algorithm works on flowgraphs. In the following, by *complete path* we mean a path in a flowgraph leading from an initial to

To appear at ISSTA'12.

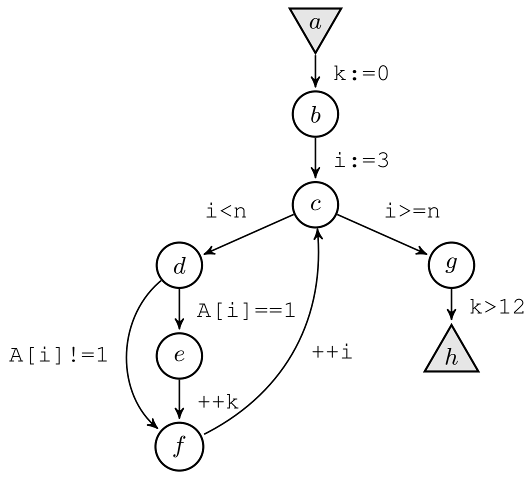

Figure 1: Flowgraph of the running example.

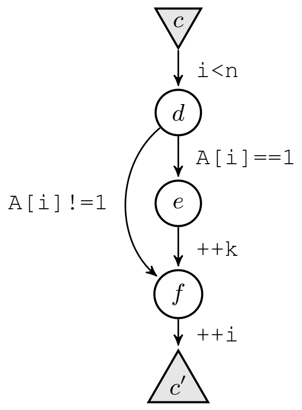

Figure 2: Flowgraph $P(\{c,d,e,f\},c)$ induced by the loop $\{c,d,e,f\}$ with entry node c along the backbone $\pi = abcgh$ of the flowgraph at Figure 1.

a target location. If a given flowgraph contains only finitely many complete paths $\rho_1,\ldots,\rho_n$, one can compute a necessary (and sufficient) condition very easily: using symbolic execution, we compute a path condition φ_i for each complete path ρ_i and we construct the necessary condition as

$$\hat{\varphi} = \bigvee_{1\le i\le n} \varphi_i.$$

Unfortunately, the number of complete paths is usually very large or even infinite as flowgraphs typically contain cycles. However, the number of acyclic complete paths is always finite and typically moderate. Therefore, we associate to each complete path ρ one acyclic complete path called the *backbone* of ρ. The backbone is defined by the following procedure: If ρ is acyclic, then the backbone is directly ρ. Otherwise, we find the leftmost repeating node in ρ, remove the part of ρ between the first and the last occurrence of this node (including the last occurrence), and repeat the procedure. In other words, the backbone arises from ρ by removing all cycles. Note that the cycles correspond to iterations in program loops.

The algorithm finds all backbones in a given flowgraph. For each backbone π, it computes an *abstract path condition*. This path condition is called abstract as it represents not only the backbone, but all complete paths with backbone π. More precisely, the abstract path condition is implied by each path condition corresponding to a complete path with backbone π. The resulting necessary condition for reaching the target location is disjunction of all abstract path conditions.

We demonstrate the computation of an abstract path condition for a backbone on our running example. All complete paths in the flowgraph of Figure 1 have the same backbone, namely $\pi = abcgh$. The crucial step in the computation of the abstract path condition for the backbone is to compute *loop summaries* of all program loops along it. A complete path with the backbone π may enter a program loop only in node c. The node c is called a *loop entry node* of the backbone π, and the corresponding program loop (given by nodes $\{c,d,e,f\}$) is a *loop along π at c*. In the following two subsections we illustrate how we compute a loop summary for the loop along π and then how we use it to compute the resulting abstract path condition. Since we use several new terms and symbols throughout these sections, we put their informal definitions into Table 1 for fast orientation in the text. We present their formal definitions later in Section 3.

2.1 Computation of a Loop Summary

The computation of a loop summary is based on an analysis of paths going around the loop, from its entry node back to it. The analysis is performed on a smaller flowgraph induced by the loop and its entry node. The flowgraph induced by the loop $\{c,d,e,f\}$ with entry node c of our running example is on Figure 2. Each iteration of the loop corresponds to one complete path in the induced flowgraph. The induced flowgraph of Figure 2 contains two complete paths, namely $\rho_1 = cdefc'$ and $\rho_2 = cdfc'$. Note that the paths ρ_1 and ρ_2 are acyclic and thus they are identical to two backbones $\pi_1 = cdefc'$ and $\pi_2 = cdfc'$ of the induced flowgraph respectively.

A loop summary consists of two parts. The first part of a loop summary is a description of an overall effect of the loop on variable values, since the first visit of the loop entry node to the last visit of the node. The effect is described by an *iterated symbolic memory*, which is a function that assigns to each program variable its value given by an expression over *symbols* and *path counters*. Symbols represent initial values of variables at the first visit of the entry node. For each variable `a`, we denote the corresponding symbol by $\underline{a}$. Path counters $\kappa_1,\kappa_2,\ldots$ correspond to different backbones of the flowgraph induced by the loop. Each path counter represents a number of loop iterations along the corresponding backbone.

In our example, we assign path counters κ_1,κ_2 to backbones π_1,π_2 respectively. Note that the backbones change only values of variables `k` and `i`. While the value of `k` is increased by one in iterations along π_1 only, the value of `i` is increased by one in each iteration along π_1 or π_2. Hence, after κ_1 iterations of π_1 and κ_2 iterations of π_2, in arbitrary order, the values of `k` and `i` are increased by κ_1 and $\kappa_1+\kappa_2$, respectively. Formally, the overall effect of the loop can be described by the iterated symbolic memory $\theta^{\vec{\kappa}}$ with only two interesting values (as the other variables are not changed in the loop):

$$\theta^{\vec{\kappa}}(\mathtt{k}) = \kappa_1 + \underline{k} \qquad \theta^{\vec{\kappa}}(\mathtt{i}) = \kappa_1 + \kappa_2 + \underline{i}$$

The vector $\vec{\kappa} = (\kappa_1,\kappa_2)$ denotes the path counters parametrising the memory $\theta^{\vec{\kappa}}$. The symbols $\underline{k}$ and $\underline{i}$ represent values of the variables `k` and `i` respectively, just before an execution of the original program enters the loop.

Table 1: General terms and symbols used throughout Section 2.

$\hat{\varphi}$	A resulting abstract path condition representing a necessary condition for reaching a target location in a given flowgraph.
π_i	An i–th backbone in an (induced) flowgraph. It is an acyclic path form the start to the target location.
κ_i	A path counter counting a number of loop iterations with the backbone π_i of the flowgraph induced by the loop and its entry node.
τ_i	A path counter representing an actual number of loop iterations with the backbone π_i of the flowgraph induced by the loop and its entry node.
$\underline{a}$	A symbol corresponding to the initial value of variable a.
$\theta^{\vec{\kappa}}$	An iterated symbolic memory parametrised by a path counters $\vec{\kappa} = (\kappa_1, \ldots, \kappa_n)$.
$\varphi^{\vec{\kappa}}$	A looping condition parametrised by a path counters $\vec{\kappa} = (\kappa_1, \ldots, \kappa_n)$.
φ'_i	An abstract path condition for the backbone π_i.
γ_i	A formula that has to be satisfied by each iteration of with backbone π_i preceded by τ_j iterations of π_j, for all j.
ψ_i	A necessary condition to perform κ_i iterations with backbone π_i. It is expressed in terms of γ_i.
$\theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}]$	The symbolic memory $\theta^{\vec{\kappa}}$, where each occurrence of each path counter $\kappa_i \in \vec{\kappa}$ has been replaced by the corresponding path counter $\tau_i \in \vec{\tau}$.
$\theta\langle\psi\rangle$	The symbolic expression ψ, where all occurrences of each symbol $\underline{a}$ in ψ have been simultaneously replaced by symbolic values $\theta(\mathtt{a})$.

Example specific terms

π	The backbone *abcgh* of the flowgraph at Figure 1.
π_1, π_2	The backbones *cdefc'*, *cdfc'* of the induced flowgraph at Figure 2, respectively.

The second part of a loop summary is a *looping condition* $\varphi^{\vec{\kappa}}$. In general, given path counters $\kappa_1, \ldots, \kappa_n$ corresponding to backbones $\pi_1, \ldots, \pi_n$ of the flowgraph induced by the loop, the formula $\varphi^{\vec{\kappa}}$ describes a necessary condition to perform $\kappa_1 + \cdots + \kappa_n$ iterations of the loop such that, for each i, exactly κ_i iterations have the backbone π_i. In other words, $\varphi^{\vec{\kappa}}$ has to be implied by each path condition φ corresponding to such $\kappa_1 + \cdots + \kappa_n$ iterations of the loop.

A looping condition is constructed in the following way. First, we compute an abstract path condition φ'_i for each backbone π_i of the flowgraph induced by the loop. Recall that φ'_i is implied by each path condition corresponding to a complete path with backbone π_i. In our running example, abstract path conditions φ'_1 and φ'_2 directly coincide with standard path conditions for backbones π_1 and π_2 as there is no loop along these backbones. Hence, the formulae φ'_1 and φ'_2 are defined as follows:

$$\begin{aligned} \varphi'_1 &\equiv \underline{i} < \underline{n} \ \wedge\ \underline{A}(\underline{i}) = 1 \\ \varphi'_2 &\equiv \underline{i} < \underline{n} \ \wedge\ \underline{A}(\underline{i}) \neq 1 \end{aligned}$$

A condition φ'_i has to be satisfied by each loop iteration with backbone π_i, where the symbols in φ'_i refer to values of corresponding variables before the iteration starts. With iterated symbolic memory $\theta^{\vec{\kappa}}$, these values can be expressed using symbols representing variable values before the first loop iteration. More precisely, after τ_i iterations with backbone π_i for each i, the values of variables are given by $\theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}]$, which is the function $\theta^{\vec{\kappa}}$ where all occurrences of κ_i is replaced by τ_i. By replacing each symbol $\underline{a}$ in φ'_i with $\theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}](\mathtt{a})$, we get a formula γ_i that has to be satisfied by each loop iteration with backbone π_i preceded by τ_1 iterations with backbone π_1, τ_2 iterations with backbone π_2, etc. In our running example, we have

$$\theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}](\mathtt{i}) = \tau_1 + \tau_2 + \underline{i} \qquad \theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}](\mathtt{n}) = \underline{n} \qquad \theta^{\vec{\kappa}}[\vec{\kappa}/\vec{\tau}](\mathtt{A}) = \underline{A}$$

and γ_1, γ_2 are thus defined as follows:

$$\begin{aligned} \gamma_1 &\equiv \tau_1 + \tau_2 + \underline{i} < \underline{n} \ \wedge\ \underline{A}(\tau_1 + \tau_2 + \underline{i}) = 1 \\ \gamma_2 &\equiv \tau_1 + \tau_2 + \underline{i} < \underline{n} \ \wedge\ \underline{A}(\tau_1 + \tau_2 + \underline{i}) \neq 1 \end{aligned}$$

Note that $\underline{i}, \underline{n}, \underline{A}$ in γ_1, γ_2 refer to values of corresponding variables before the first loop iteration.

Using γ_i we build a formula ψ_i that is a necessary condition to perform κ_i loop iterations with backbone π_i. We simply say that for each τ_i satisfying $0 \leq \tau_i < \kappa_i$, there exists a configuration of iteration counters $\vec{\tau_i} = (\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots \tau_n)$ such that γ_i holds. Moreover, each iteration counter τ_i of $\vec{\tau_i}$ has to be valid, i.e. it satisfies $0 \leq \tau_j \leq \kappa_j$. Hence, we set

$$\psi_i \equiv \forall \tau_i \ (0 \leq \tau_i < \kappa_i \rightarrow \exists \vec{\tau_i} \ (\vec{0} \leq \vec{\tau_i} \leq \vec{\kappa_i} \wedge \gamma_i)),$$

where $\vec{\kappa_i} = (\kappa_1, \ldots, \kappa_{i-1}, \kappa_{i+1}, \ldots \kappa_n)$ and $\vec{0} \leq \vec{\tau_i} \leq \vec{\kappa_i}$ is an abbreviation of a conjunction of all formulae $0 \leq \tau_j \leq \kappa_j$ such that $1 \leq j \leq n, j \neq i$. The overall looping condition $\varphi^{\vec{\kappa}}$ is now defined as a conjunction $\psi_1 \wedge \ldots \wedge \psi_n$. In our running example, we get the following looping condition:

$$\begin{aligned} \varphi^{\vec{\kappa}} &\equiv \psi_1 \wedge \psi_2 \\ \psi_1 &\equiv \forall \tau_1 \big(0 \leq \tau_1 < \kappa_1 \rightarrow \exists \tau_2 (0 \leq \tau_2 \leq \kappa_2 \wedge \\ &\qquad \wedge \tau_1 + \tau_2 + \underline{i} < \underline{n} \ \wedge\ \underline{A}(\tau_1 + \tau_2 + \underline{i}) = 1)\big) \\ \psi_2 &\equiv \forall \tau_2 \big(0 \leq \tau_2 < \kappa_2 \rightarrow \exists \tau_1 (0 \leq \tau_1 \leq \kappa_1 \wedge \\ &\qquad \wedge \tau_1 + \tau_2 + \underline{i} < \underline{n} \ \wedge\ \underline{A}(\tau_1 + \tau_2 + \underline{i}) \neq 1)\big) \end{aligned}$$

2.2 Computation of Abstract Path Conditions

We assign computed loop summaries to the corresponding loop entry nodes on a backbone of the original flowgraph and we apply (a slight modification of) symbolic execution to get an abstract path condition for the backbone.

In our running example, we first assign the computed summary $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}})$ to the loop entry c of the backbone $\pi = abcgh$ of the flowgraph of Figure 1, and then we execute the backbone π symbolically as follows. The first two edges of the backbone are executed in the standard way. We get a symbolic memory θ_1 with only two interesting values (the other variables are not changed in our program and hence we do not track them)

$$\theta_1(\mathtt{k}) = 0 \qquad\qquad \theta_1(\mathtt{i}) = 3$$

and a path condition $\varphi_1 \equiv true$. As we are now in the loop entry node c, we apply the loop summary $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}})$ computed before. We need to compose θ_1 and φ_1 with the summary to achieve a symbolic memory θ_2 and a path condition φ_2 representing variable values and a path condition after the loop. Let us first discuss the computation of θ_2.

We know that $\theta_1(\mathtt{i}) = 3$ and $\theta^{\vec{\kappa}}(\mathtt{i}) = \kappa_1 + \kappa_2 + \underline{i}$. The symbol $\underline{i}$ represents the value of the variable i when we reach the node c for the first time. This value is given by $\theta_1(\mathtt{i})$. Therefore, $\theta_2(\mathtt{i}) = \kappa_1 + \kappa_2 + \theta_1(\mathtt{i}) = \kappa_1 + \kappa_2 + 3$. In general, for each variable v, $\theta_2(\mathtt{v})$ is given by $\theta^{\vec{\kappa}}(\mathtt{v})$ where all symbols $\underline{a}$ were simultaneously replaced by corresponding symbolic values $\theta_1(\mathtt{a})$. We can summarise the content of θ_2 as follows:

$$\theta_2(\mathtt{k}) = \kappa_1 \qquad \theta_2(\mathtt{i}) = \kappa_1 + \kappa_2 + 3$$

Now we focus on the computation of φ_2. The formula $\varphi^{\vec{\kappa}}$ contains symbols $\underline{n}, \underline{i}, \underline{A}$ representing values before the loop. Again, we replace these by corresponding values in θ_1. This substitution has exactly the same meaning as in the computation of the value $\theta_2(\mathtt{i})$ discussed before. We get an abstracted path condition $\varphi_2 \equiv \varphi_1 \wedge \theta_1\langle\varphi^{\vec{\kappa}}\rangle$, where $\theta_1\langle\varphi^{\vec{\kappa}}\rangle$ is the formula $\varphi^{\vec{\kappa}}$ with all the symbols $\underline{a}$ appearing in it simultaneously replaced by corresponding symbolic values $\theta_1(\mathtt{a})$. Since $\varphi_1 \equiv true$, we get $\varphi_2 \equiv \theta_1\langle\varphi^{\vec{\kappa}}\rangle$.

At the end, we process the last two edges of the backbone $\pi = abcgh$. The edges do not change variable values, but they extend the abstracted path condition φ_2 with tests `i>=n` and `k>12` evaluated in the abstract symbolic memory θ_2. Hence, we get an abstracted path condition

$$\varphi_3 \equiv \theta_1\langle\varphi^{\vec{\kappa}}\rangle \;\wedge\; \kappa_1 + \kappa_2 + 3 \geq \underline{n} \;\wedge\; \kappa_1 > 12.$$

To obtain the resulting abstract path condition for the backbone, we existentially quantify all path counters with a free occurrence in the formula φ_3 and we state that values of the path counters have to be non-negative. Hence, the resulting abstract path condition for the backbone $\pi = abcgh$ of our example is

$$\varphi_4 \equiv \exists\kappa_1, \kappa_2\,(\,\kappa_1, \kappa_2 \geq 0 \;\wedge\; \theta_1\langle\varphi^{\vec{\kappa}}\rangle \;\wedge \\ \wedge\; \kappa_1 + \kappa_2 + 3 \geq \underline{n} \;\wedge\; \kappa_1 > 12\,).$$

The necessary condition $\hat{\varphi}$ is then a disjunction of all abstract path conditions for backbones of the analysed flowgraph. As there is only one backbone $\pi = abcgh$ in our running example, we directly get $\hat{\varphi} \equiv \varphi_4$.

2.3 Imprecision of Abstract Path Conditions

In general, the formula $\hat{\varphi}$ is not a sufficient condition on inputs to reach the target node. There are basically two sources of this imprecision.

- It is not always possible to express the overall effect of a loop on a variable in the presented way. Moreover, our algorithm may fail to express it even if it is expressible. In such cases, the variable is assigned the special value $\star$ with the meaning "unknown". If we symbolically execute a test containing a variable with the value $\star$, we do not add a result of this test to our path condition.

- The looping condition is constructed as a necessary but not a sufficient condition. More precisely, it checks whether tests in each iteration are satisfied for the iterated symbolic memory with some admissible values of path counters, but the consistency of these admissible values over all iterations is not checked.

3. DESCRIPTION OF THE ALGORITHM

For simplicity, we describe our algorithm for programs manipulating only integer variables and read-only multidimensional integer arrays, and with no function calls. Nevertheless, the algorithm can be extended to handle variables of other types, function calls, etc.

3.1 Preliminaries

A flowgraph is a tuple $P = (V, E, l_s, l_t, \iota)$, where (V, E) is a finite oriented graph, $l_s, l_t \in V$ are different *start* and *target nodes*

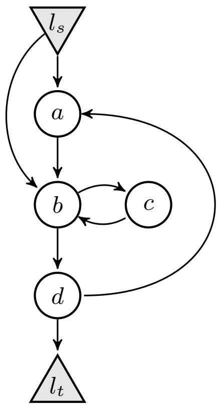

Figure 3: Flowgraph with more loops.

respectively, in-degree of l_s is 0, and $\iota : E \to \mathcal{I}$ is a function assigning to each edge e an *instruction* $\iota(e)$. A node is *branching* if its out-degree is 2. All other nodes have out-degree at most 1. We use two kinds of instruction: an assignment instruction $\mathtt{a}$:=e for some scalar variable $\mathtt{a}$ and some expression e, and an assumption $\mathtt{assume}(\gamma)$ for some quantifier-free formula γ over program variables. Out-edges of any branching node are labelled with assumptions $\mathtt{assume}(\gamma)$ and $\mathtt{assume}(\neg\gamma)$ for some γ. We often omit the keyword `assume` in our flowgraphs. We consider only instructions operating on *scalar variables* $\mathtt{a}, \mathtt{b}, \ldots$ of type `Int` and multi-dimensional *array variables* $\mathtt{A}, \mathtt{B}, \ldots$ of type $\mathtt{Int}^k \to \mathtt{Int}$. Note that we often identify programs with the corresponding flowgraphs.

A *path* in a flowgraph is a finite sequence $v_1 v_2 \cdots v_k$ of nodes such that $k \geq 0$ and $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. Paths are always denoted by Greek letters. The terms *complete path* and its *backbone* have already been defined in the previous section.

Now we formalise definitions of loops and loop entry nodes on a backbone. Let π be a backbone with a prefix αv. There is a *loop* C with an *entry node* v along π, if there exists a path $v\beta v$ such that no node of β appears in α. The loop C is then the smallest set containing all nodes of all such paths $v\beta v$. For example, the flowgraph of Figure 3 contains two backbones: $\pi_1 = l_s bdl_t$ and $\pi_2 = l_s abdl_t$. While π_1 contains only one loop $\{a, b, c, d\}$ with entry node b, π_2 contains loop $\{a, b, c, d\}$ with entry node a and loop $\{b, c\}$ with entry node b.

For a loop C with an entry node v, a *flowgraph induced by the loop*, denoted as $P(C, v)$, is the subgraph of the original flowgraph induced by C where v is marked as the start node, a fresh node v' is added and marked as the target node, and every edge $(u, v) \in E$ leading to v is replaced by an edge (u, v') labelled with the same instruction as one of the edge (u, v).

By *symbolic expressions* we mean all expressions build with integers, standard integer operations and functions, and

- a *symbol* $\underline{a}$ for each scalar variable $\mathtt{a}$,
- a *function symbol* $\underline{A}$ for each array variable $\mathtt{A}$, where arity of $\underline{A}$ corresponds to the dimension of array $\mathtt{A}$,
- a countable set $\{\kappa_1, \tau_1, \kappa_2, \tau_2, \ldots\}$ of variables called *path counters*, and
- a construct $\mathbf{ite}(\varphi, e_1, e_2)$, of meaning "**i**f-**t**hen-**e**lse", where e_1, e_2 are symbolic expressions and φ is a first order formula over symbols and path counters. The value of $\mathbf{ite}(\varphi, e_1, e_2)$ is e_1 if φ holds and e_2 otherwise. And finally,

- a special symbol $\star$ called *unknown*.

Let $f, e_1, \ldots, e_n$ be symbolic expressions and $x_1, \ldots, x_n$ be path counters or symbols corresponding to scalar variables. Then $f[x_1/e_1, \ldots, x_n/e_n]$ is a symbolic expression f where all occurrences of all x_i are simultaneously replaced by expressions e_i. To shorten the notation, we write $f[\vec{x}/\vec{e}]$ when the meaning is clearly given by a context. We also use the notation $\varphi[\vec{x}/\vec{e}]$ with the analogous meaning for formulae φ.

A *symbolic memory* is a function θ assigning to each scalar variable `a` a symbolic expression and to each array variable `A` the function symbol $\underline{A}$. Let `a` be a scalar variable and e be a symbolic expression. Then $\theta[\mathtt{a} \to e]$ is a symbolic memory equal to θ except for variable `a`, where $\theta[\mathtt{a} \to e](\mathtt{a}) = e$. The notation $\theta(\cdot)$ is used in a more general way. It always denotes the operation of replacing each (scalar or array) variable `a` by $\theta(\mathtt{a})$. $\theta\langle\cdot\rangle$ denotes the operation on symbolic expressions or formulae, where all occurrences of all symbols $\underline{a}$ are simultaneously replaced by $\theta(\mathtt{a})$. Additionally, $\theta_1\langle\theta_2\rangle$ denotes composition of two symbolic memories θ_1 and θ_2 satisfying $\theta_1\langle\theta_2\rangle(\mathtt{a}) = \theta_1\langle\theta_2(\mathtt{a})\rangle$ for each variable `a`. Intuitively, the symbolic memory $\theta_1\langle\theta_2\rangle$ represents an overall effect of a code with effect θ_1 followed by a code with effect θ_2.

Finally, for vectors $\vec{u} = (u_1, \ldots, u_n)$ and $\vec{v} = (v_1, \ldots, v_n)$ we use $\vec{u} \leq \vec{v}$ as an abbreviation for $u_1 \leq v_1 \wedge \ldots \wedge u_n \leq v_n$.

3.2 The Algorithm

Now we precisely formulate our algorithm computing a necessary condition $\hat{\varphi}$ for reaching a target node of a given flowgraph. We provide more details, which includes dealing with nested loops.

To compute the necessary condition, we call Algorithm 1 on the set of all backbones $\{\pi_1, \ldots, \pi_k\}$ of the given program. The algorithm performs a modified symbolic execution of these backbones described later. We extract abstract path conditions $\varphi_1, \ldots, \varphi_k$ from the algorithm's output and we compute the necessary condition as

$$\hat{\varphi} \equiv \bigvee_{1 \leq i \leq k} \exists \vec{\kappa}_i \; (\, \vec{\kappa}_i \geq 0 \;\wedge\; \varphi_i \,),$$

where $\vec{\kappa}_i$ is a vector of all path counters having a free occurrence in φ_i.

The Algorithm 1 symbolically executes backbones one by one (see line 2). For each backbone π_i, we first analyse all loops along it (see **foreach** loop at lines 3–9). Since we convert loops into induced flowgraphs (see lines 4–6), we can analyse loops (and their nested loops) in the same way as we analyse the original program (see line 7). The main part of the loop analysis is the computation of loop summaries (see line 8) performed by Algorithm 2, which we discuss later. After analysis of loops along π_i, the backbone is symbolically executed (see initialisation at lines 10–12 and **foreach** loop at lines 13–22). The symbolic execution differs from the original one only at loop entry locations, where we process the computed loop summaries. In both lines 20 and 22 there we apply symbol-substitution operations $\cdot\langle\cdot\rangle$. The purpose of these operations is to compose symbolic execution up to the loop entry vertex with symbolic execution of the loop, over-approximated by the summary. For each backbone π_i, the algorithm produces an abstracted symbolic memory θ_i and an abstracted path condition φ_i, which are added into the `result` at line 23.

3.3 Computation of a Loop Summary

The computation is depicted in Algorithm 2. The algorithm first introduces new path counters $\vec{\kappa}$ for backbones within the loop. Note that the algorithm knows the effect of each backbone within the loop as it gets the corresponding symbolic memories and abstract

Algorithm 1: executeBackbones($\{\pi_1, \ldots, \pi_k\}$)

```
Input:
    {$\pi_1, \ldots, \pi_k$} // backbone paths to be executed
Output:
    {$(\pi_1, \theta_1, \varphi_1), \ldots, (\pi_k, \theta_k, \varphi_k)$}
                // result of symbolic execution of backbones
1  result $\longleftarrow \emptyset$
2  foreach $i = 1, \ldots, k$ do
3      foreach loop entry $v$ along $\pi_i$ do
4          Let $C$ be the loop at $v$
5          Compute induced flowgraph $P(C, v)$ for $C$ at $v$
6          Let $\{\pi'_1, \ldots, \pi'_l\}$ be all backbones in $P(C, v)$
7          $\{(\pi'_1, \theta'_1, \varphi'_1), \ldots, (\pi'_l, \theta'_l, \varphi'_l)\} \longleftarrow$
               executeBackbones( $\{\pi'_1, \ldots, \pi'_l\}$)
8          $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}}) \longleftarrow$ computeSummary(
                   $\{(\pi'_1, \theta'_1, \varphi'_1), \ldots, (\pi'_l, \theta'_l, \varphi'_l)\}$)
9          Attach the summary $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}})$ to the loop entry $v$
10     Initialise $\theta_i$ to return $\underline{a}$ for each variable a
11     $\varphi_i \longleftarrow$ true
12     Let $\pi_i = v_1 \ldots v_n$
13     foreach $j = 1, \ldots, n-1$ do
14         if $\iota((v_j, v_{j+1}))$ has the form assume($\gamma$) and $\theta_i(\gamma)$
           contains no $\star$ then
15             $\varphi_i \longleftarrow \varphi_i \wedge \theta_i(\gamma)$
16         if $\iota((v_j, v_{j+1}))$ has the form a $\longleftarrow e$ then
17             $\theta_i \longleftarrow \theta_i[\mathtt{a} \to \theta_i(e)]$
18         if $v_{j+1}$ is a loop entry then
19             Let $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}})$ be the summary attached to $v_{j+1}$
20             $\varphi_i \longleftarrow \varphi_i \wedge \theta_i\langle\varphi^{\vec{\kappa}}\rangle$
21             Replace all predicates of $\varphi_i$ containing $\star$ by true
22             $\theta_i \longleftarrow \theta_i\langle\theta^{\vec{\kappa}}\rangle$
23     Insert triple $(\pi_i, \theta_i, \varphi_i)$ into result
24 return result
```

path conditions as input. The only task is to combine these symbolic memories into an iterated symbolic memory $\theta^{\vec{\kappa}}$ and to assemble a looping condition $\varphi^{\vec{\kappa}}$.

The first half of the algorithm (see lines 2-10) computes the iterated memory $\theta^{\vec{\kappa}}$. To be on the safe side, we start with $\theta^{\vec{\kappa}}$ assigning $\star$ to all scalar variables. Then we gradually improve the precision of $\theta^{\vec{\kappa}}$. The crucial step is the computation of an improved value e for a scalar variable `a` (see line 6). The value e is defined as $\star$ except in the following cases:

1. For each backbone π'_i, we have $\theta'_i(\mathtt{a}) = \underline{a}$. In other words, the value of `a` is not changed in any iteration of the loop. This case is trivial. We set $e = \underline{a}$.

2. For each backbone π'_i, either $\theta'_i(\mathtt{a}) = \underline{a}$ or $\theta'_i(\mathtt{a}) = \underline{a} + d_i$ for some symbolic expression d_i such that $\theta^{\vec{\kappa}}\langle d_i\rangle$ contains neither $\star$ nor any path counters. Let us assume that the latter possibility holds for backbones $\pi'_1, \ldots, \pi'_m$. The condition on $\theta^{\vec{\kappa}}\langle d_i\rangle$ guarantees that the value of d_i is constant during all iterations over the loop. In this case, we set $e = \underline{a} + \sum_{1 \leq i \leq m} \theta^{\vec{\kappa}}\langle d_i\rangle \cdot \kappa_i$.

3. There exists a symbolic expression d such that $\theta^{\vec{\kappa}}\langle d\rangle$ contains neither $\star$ nor any path counters. For each backbone π'_i, either $\theta'_i(\mathtt{a}) = \underline{a}$ or $\theta'_i(\mathtt{a}) = d$. Let us assume that the latter possibility holds for backbones $\pi'_1, \ldots, \pi'_m$. In other words, the value of `a` is set to d in each iteration with backbone π'_i

Algorithm 2: computeSummary($\{(\pi'_1,\theta'_1,\varphi'_1)$,...,$(\pi'_l,\theta'_l,\varphi'_l)\}$)

```
Input:
    {(π'_1, θ'_1, φ'_1), ..., (π'_l, θ'_l, φ'_l)}
                    // results from single execution of backbones
Output:
    (θ^κ⃗, φ^κ⃗) // the computed summary
1  Introduce fresh path counters $\vec{\kappa} = (\kappa_1,\ldots,\kappa_l)$ for backbones
   $\pi'_1,\ldots,\pi'_l$ respectively
2  Initialise $\theta^{\vec{\kappa}}$ to return $\star$ for each scalar variable
3  repeat
4      change ⟵ false
5      foreach scalar variable a do
6          Compute an improved value $e$ for the variable a from
           symbolic memories $\theta'_1,\ldots,\theta'_l$ and $\theta^{\vec{\kappa}}$
7          if $e \neq \star \wedge \theta^{\vec{\kappa}}(\mathtt{a}) = \star$ then
8              $\theta^{\vec{\kappa}} \longleftarrow \theta^{\vec{\kappa}}[\mathtt{a} \to e]$
9              change ⟵ true
10 until change = false
11 foreach $i = 1,\ldots,l$ do
12     Let $\vec{\kappa}'_i$ be a vector of all path counters having a free
       occurrence in $\varphi'_i$
13     $\gamma'_i \longleftarrow \theta^{\vec{\kappa}}\langle\varphi'_i\rangle[\vec{\kappa}/\vec{\tau}]$, where $\vec{\tau} = (\tau_1,\ldots,\tau_l)$
14     Replace all predicates of $\gamma'_i$ containing $\star$ by true
15     Let $\vec{\kappa}_i = (\kappa_1,\ldots,\kappa_{i-1},\kappa_{i+1},\ldots\kappa_l)$ and
           $\vec{\tau}_i = (\tau_1,\ldots,\tau_{i-1},\tau_{i+1},\ldots\tau_l)$
16     $\psi'_i \longleftarrow \forall\tau_i\,(0 \leq \tau_i < \kappa_i \to$
               $\exists\vec{\tau}_i\,(\vec{0} \leq \vec{\tau}_i \leq \vec{\kappa}_i \wedge \exists\vec{\kappa}'_i(\vec{\kappa}'_i \geq 0 \wedge \gamma'_i)))$
17 $\varphi^{\vec{\kappa}} \longleftarrow \psi'_1 \wedge \ldots \wedge \psi'_l$
18 return $(\theta^{\vec{\kappa}}, \varphi^{\vec{\kappa}})$
```

for $1 \leq i \leq m$, while it is unchanged in any other iteration. Hence, we set $e = \mathbf{ite}(\sum_{1\leq i\leq m}\kappa_i > 0, \theta^{\vec{\kappa}}\langle d\rangle, \underline{a})$.

4. For one backbone, say π'_i, $\theta'_i(\mathtt{a}) = d$ for some symbolic expression d such that $\theta^{\vec{\kappa}}\langle d\rangle$ contains neither $\star$ nor any path counters except κ_i. Further, for each backbone π'_j such that $i \neq j$, $\theta'_j(\mathtt{a}) = \underline{a}$. That is, only iterations with backbone π'_i modify a and they set it to a value independent on other path counters than κ_i. Note that if we assign d to a in the κ_i-th iteration with backbone π'_i, then the actual assigned value of d is the value after $\kappa_i - 1$ iterations along the paths. Therefore we set $e = \mathbf{ite}(\kappa_i > 0, (\theta^{\vec{\kappa}}\langle d\rangle)[\kappa_i/\kappa_i - 1], \underline{a})$.

Note that one can add another cases covering other situations where the value of a can be expressed precisely, e.g. the case capturing geometric progressions. Also note that symbolic value e computed by the rules represent precise value for the analysed variable a. Therefore, a value of any variable in the resulting iterated symbolic memory $\theta^{\vec{\kappa}}$ is expressed either precisely, or it is completely unknown (i.e. $\star$). We may consider abstract values in between the corner cases, but we leave this for future research.

The second half of the algorithm (see lines 11-17) builds looping condition $\varphi^{\vec{\kappa}}$. The **foreach** loop at lines 11–16 computes for each backbone π'_i the formula ψ'_i. There are several lines which exactly match the description we gave in Section 2.1. First of all, we can recognise the computation of formula γ'_i at line 13. At line 15, there we introduce vectors $\vec{\kappa}_i$ and $\vec{\tau}_i$ of path counters, which we use at line 16 to express the formula ψ'_i. But we can notice, that the formula ψ'_i now contains a sub-formula $\exists\vec{\kappa}'_i(\vec{\kappa}'_i \geq 0 \wedge \gamma'_i)$ instead of simply γ'_i. This is because there can be some nested loops along the backbone π'_i. Therefore, φ'_i may contain free occurrences of path counters introduced in loop summaries of these nested loops. We collect these path counters in vector $\vec{\kappa}'_i$ at line 12. We have to existentially bind these path counters in the formula ψ'_i, since for each τ_i-th execution of the backbone π'_i there can be different values of the path counters $\vec{\kappa}'_i$ of the loops along π'_i. We have already discussed all the lines of the **foreach** loop, except line 14. Note that some variables in the iterated symbolic memory $\theta^{\vec{\kappa}}$ may have the unknown value $\star$. Then the computation at line 13 may bring some of these values $\star$ into the resulting formula γ'_i. Predicates containing values $\star$ are useless. Therefore, we remove them. This in turn also weakens ψ'_i. And just below the **foreach** loop at line 17, there we construct the resulting looping condition $\varphi^{\vec{\kappa}}$ as the conjunction of the formulae ψ'_i computed in the loop above. This construction matches the intuition we gave in Section 2.1.

3.4 Loss of Precision Caused by Nested Loops

Iterated symbolic memory $\theta^{\vec{\kappa}}$ computed by the Algorithm 2 can lose some precision also due to nested loops. We illustrate it on the following program.

```
for (i = 0; i < m; ++i) {
  j = i;
  while (j < n)
    ++j;
}
```

The corresponding flowgraph is depicted in Figure 4 (top). The flowgraph contains one backbone $l_s a l_t$ with the loop entry node a and the corresponding loop $C = \{a,b,c,d,e\}$. The induced flowgraph $P' = P(C,a)$ contains again one backbone $abcea'$ with entry node c and the corresponding loop $C' = \{c,d\}$. The induced flowgraphs P' and $P'' = P(C',c)$ are depicted in Figure 4 (bottom left and bottom right respectively). Note that P'' has a single backbone cdc'.

If we symbolically execute the backbone cdc' of the induced program P'' by the Algorithm 1, we get the following symbolic memory θ'' and path condition φ'':

$$\theta''(\mathtt{j}) = \underline{j} + 1 \qquad \varphi'' \equiv \underline{j} < \underline{n}$$
$$\theta''(\mathtt{n}) = \underline{n}$$

Now we apply Algorithm 2 on $\{(cdc',\theta'',\varphi'')\}$ to compute the loop summary $(\theta^{\kappa'},\varphi^{\kappa'})$ for the loop $\{c,d\}$ with entry node c along the backbone $abcea'$ of the induced program P', we get the following:

$$\theta^{\kappa'}(\mathtt{j}) = \underline{j} + \kappa' \qquad \varphi^{\kappa'} \equiv \forall\,\tau'(0 \leq \tau' < \kappa' \;\to\; \underline{j} + \tau' < \underline{n})$$
$$\theta^{\kappa'}(\mathtt{n}) = \underline{n}$$

This loop summary is used in symbolic execution of the backbone $abcea'$ of the induced program P' performed again by Algorithm 1. The symbolic execution produces the following symbolic memory θ' and abstract path condition φ':

$$\begin{array}{ll} \theta'(\mathtt{i}) = \underline{i} + 1 & \varphi' \equiv \underline{i} < \underline{m} \wedge \\ \theta'(\mathtt{j}) = \underline{i} + \kappa' & \quad \forall\,\tau'(0 \leq \tau' < \kappa' \to \underline{i} + \tau' < \underline{n}) \wedge \\ \theta'(\mathtt{m}) = \underline{m} & \quad \underline{i} + \kappa' \geq \underline{n} \\ \theta'(\mathtt{n}) = \underline{n} & \end{array}$$

And finally, we run Algorithm 2 with the triple $\{(abcea',\theta',\varphi')\}$ to get the loop summary for the loop $\{a,b,c,d,e\}$ with entry node a along the backbone $l_s a l_t$ of the program P. We get the following iterated symbolic memory θ^{κ}:

$$\begin{array}{ll} \theta^{\kappa}(\mathtt{i}) = \underline{i} + \kappa & \theta^{\kappa}(\mathtt{m}) = \underline{m} \\ \theta^{\kappa}(\mathtt{j}) = \star & \theta^{\kappa}(\mathtt{n}) = \underline{n} \end{array}$$

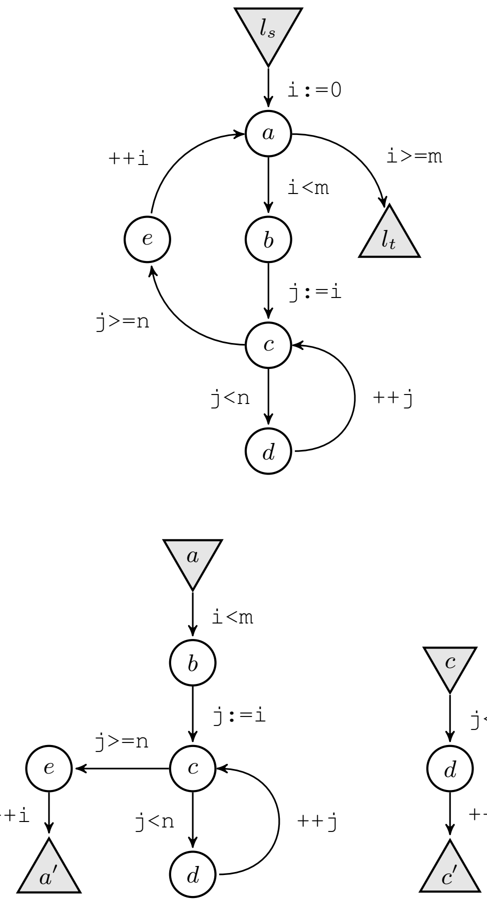

Figure 4: Example of a program P (top) with nested path counter dependency. Program P' (bottom left) is induced by loop $C = \{a, b, c, d, e\}$ with entry node a of the program P. Program P'' (bottom right) is induced by loop $C' = \{c, d\}$ with entry node c of the program P'.

In fact, the value of j after one iteration of P' can be expressed without κ' as $\theta'(\mathtt{j}) = \max(\underline{n}, \underline{i})$. If we modify θ' in this way, then the Algorithm 2 automatically computes more precise iterated symbolic memory θ^{κ}, namely for the variable j it returns

$$\theta^{\kappa}(\mathtt{j}) = \mathbf{ite}(\kappa > 0, \max(\underline{n}, \underline{i} + \kappa - 1), \underline{j}).$$

The crucial step towards higher precision of iterated symbolic memory is detection of dependencies between path counters of an outer loop and path counters of its nested loops. In the example, we would like to detect the fact that in $(\kappa + 1)$-st iteration of P', the nested loop is iterated $\kappa' = \max(0, \underline{n} - (\underline{i} + \kappa))$ times. To improve the precision, we have developed a 'heavyweight' version of Algorithm 2, which tries to find a linear relations between inner and outer path counters using an SMT solver. For details see [27].

3.5 Soundness, Incompleteness, Termination

We finish the description of the algorithm by formulating its soundness, incompleteness and termination theorems.

THEOREM 1 (SOUNDNESS). *Let $\hat{\varphi}$ be the necessary condition computed by our algorithm for a given program and a given target location in it. If $\hat{\varphi}$ is not satisfiable, then the target location is not reachable in that program.*

PROOF. *(Sketch)* Let us suppose that the target location is reachable. Then there exists a complete path σ such that the standard symbolic execution of this path produces a satisfiable path condition. We prove that this path condition implies $\hat{\varphi}$. Hence, we get satisfiability of $\hat{\varphi}$.

The formula $\hat{\varphi}$ is constructed as $\bigvee_{1 \le i \le k} \exists\vec{\kappa}_i \, (\vec{\kappa}_i \ge 0 \;\wedge\; \varphi_i)$, where each φ_i is an abstract path condition for backbone π_i. Let π_i be the backbone of σ. Hence, it is sufficient to show that the formula $\exists\vec{\kappa}_i \, (\vec{\kappa}_i \ge 0 \;\wedge\; \varphi_i)$, where φ_i is the abstract path condition for the backbone π_i, is implied by the path condition of the symbolic execution along σ. This can be shown by double induction on maximal loop nesting depth (top-level induction) and length of the backbone π_i (nested induction). □

THEOREM 2 (INCOMPLETENESS). *There is a program and an unreachable target location in it for which the formula $\hat{\varphi}$ computed by our algorithm is satisfiable.*

PROOF. Let us consider the following C code:

```
int i = 1;
while (i < 3)
    if (i == 2) i = 1;
    else i = 2;
assert(false);
```

The loop never terminates. Therefore, a program location below it (the assertion) is not reachable. But $\hat{\varphi}$ computed according to our algorithm for that location is equal to $true$, since value of variable i is expressed as $\star$ in the iterated symbolic memory of the loop summary. □

THEOREM 3 (TERMINATION). *Algorithm 1 executed on all the backbones of a given program P always terminates.*

PROOF. A backbone is a finite path in P, so there is a finite number of loop entries along it. Therefore, the loop at line 3 terminates if there is only finite number of recursive calls at line 7 and Algorithm 2 called at line 8 also always terminates. The function executeBackbones is recursively called on induced programs. As P is finite, there is a finite number of nested loops in each loop along any backbone. The loop at line 3 of the Algorithm 2 must always terminate, since (1) there is only finite number of variables in P, and (2) for each variable a, the line 8 can be executed at most once during the algorithm. Therefore the condition at line 7 eventually becomes always $false$. □

3.6 Limitations of the Algorithm

The main limitation of our algorithm computing necessary condition comes from the fact that it cannot compute precise effect of a loop in all cases. Let us consider the programs

```
for(...)                   for(...)
  if(i<0) j=j+1;             if(i<0) j=j+1;
  else j=j+2;                else j=j*2;
```

where the symbol '...' represents some unspecified progression of the variable i. The two programs are very similar: they differ only in the last statement. In both cases, an induced program corresponding to the loop's body has two backbones π_1 and π_2 going through the positive and the negative branch of the if-else statement respectively. We assign path counters κ_1 and κ_2 to the backbones π_1 and π_2 respectively. The difference become important when we compute an iterated symbolic memory $\theta^{\vec{\kappa}}$ for the cycle. For the left program, we easily compute that $\theta^{\vec{\kappa}}(\mathtt{j}) = \underline{j} + \kappa_1 + 2\kappa_2$, where $\underline{j}$ represents an initial value of j. For the right program, our algorithm computes $\theta^{\vec{\kappa}}$ with $\theta^{\vec{\kappa}}(\mathtt{j}) = \star$. The reason is that the resulting value of j heavily depends on interleaving of the positive and negative branches of the `if-else` statement when iterating the loop. In general, we cannot declaratively describe a value of a variable after execution of a loop if the value depends on interleaving of paths along the loop and if we cannot deduce the interleaving from any other information. This is a principal limitation of our approach.

However, in many practical cases the interleaving of paths along a loop can be derived from other information. Then our technique can be adjusted to handle such cases. For example, assume that the first line of the right program looks like

```
for( ; i<n; i+=c)
```

for some fixed integer c. Then $\theta^{\vec{\kappa}}(\mathtt{i}) = \underline{i} + c(\kappa_1 + \kappa_2)$ and the test `i<0` is evaluated into the predicate $\underline{i} + c(\kappa_1 + \kappa_2) < 0$. This allow as to express $\theta^{\vec{\kappa}}(\mathtt{j})$ precisely again as $(\underline{j} + \kappa_1)2^{\kappa_2}$ if $c \geq 0$ and as $\underline{j} + 2^{\kappa_2} + \kappa_1$ otherwise. This can be formulated as another rule for Section 3.3. To sum up, technique can be improved to handle more cases without loss of precision.

These improvements leads to another, and perhaps more important, issue: the performance of SMT solvers. Let us see this on our example. The proposed rule would produce symbolic value for $\theta^{\vec{\kappa}}(\mathtt{j})$ with the exponential function containing expression 2^{κ_2}. Since values in $\theta^{\vec{\kappa}}$ commonly appear in a resulting abstract path condition $\hat{\varphi}$, we significantly increase complexity of satisfiability queries to the SMT solver. Hence, one has to find a reasonable trade-off between precision of produced necessary conditions and time needed to solve them by available SMT solvers. In this paper we do not give an answer what composition of rules gives an optimal $\hat{\varphi}$. We only propose one, which seems to perform well according to our experimentation (see Section 6). Nevertheless, we can easily observe that the efficiency of applications of necessary conditions depends on *what* rules we choose and *not* on how many rules we can provide in total.

There is another issue related to SMT solvers. It is widely known that current SMT solvers have serious performance issues when dealing with quantified formulae. Since formulae $\hat{\varphi}$ computed by our algorithm commonly contain quantifiers, we tackle this issue in separate Section 4.

4. DEALING WITH QUANTIFIERS

We can ask an SMT solver whether a computed necessary condition $\hat{\varphi}$ is satisfiable or not. If it is, we may further ask for a model. As we will see in Section 5, such queries to a solver should be fast. Unfortunately, our experience with solvers shows that presence of quantifiers in $\hat{\varphi}$ usually causes performance issues. To overcome this issue, we introduce a transformation of $\hat{\varphi}$ into a quantifier-free formula $\hat{\varphi}^K$ that is implied by $\hat{\varphi}$ and thus remains necessary. The transformation is parametrised by $K \geq 0$.

One can immediately see that all universal quantifiers in $\hat{\varphi}$ come from formulae ψ_i' of line 16 of Algorithm 2. Each formula ψ_i' has the form

$$\psi_i' \equiv \forall \tau_i \; (0 \leq \tau_i < \kappa_i \; \rightarrow \; \rho(\tau_i)).$$

Clearly, the formula is equivalent to $\bigwedge_{0 \leq \tau_i < \kappa_i} \rho(\tau_i)$. We do not know the value of κ_i, but we can weaken the formula to check only the first K instances of $\rho(\tau_i)$. In other words, we replace each ψ_i' in $\hat{\varphi}$ by a weaker formula

$$\psi_i^K \equiv \bigwedge_{0 \leq \tau_i \leq K} (\tau_i < \kappa_i \; \rightarrow \; \rho(\tau_i)).$$

Having eliminated all universal quantifiers, we can also eliminate existential quantification of all κ_i and all $\vec{\tau_i}$ by redefining them as uninterpreted integer constants. The resulting formula is denoted as $\hat{\varphi}^K$.

Let us note that the choice of K affects the length and precision of $\hat{\varphi}^K$: the higher value of K we choose, the stronger and longer formula $\hat{\varphi}^K$ we get. Hence, higher values of K lead to longer running times of an SMT solver. However, for lower values of K, the information represented by $\hat{\varphi}^K$ can be less significant or insignificant at all. For example, one can easily find a program with an unreachable location and some $K > 0$ such that the necessary formula $\hat{\varphi}$ is unsatisfiable and $\hat{\varphi}^{K'}$ is unsatisfiable for all $K' \geq K$ and satisfiable for all $K' < K$.

In some cases, an SMT solver decides satisfiability of $\hat{\varphi}$ very quickly: even in a shorter time than needed for transformation of $\hat{\varphi}$ into $\hat{\varphi}^K$. In practice, we ask the solver for satisfiability of $\hat{\varphi}$ and, in parallel, we transform $\hat{\varphi}$ into $\hat{\varphi}^K$ and then ask the solver for satisfiability of $\hat{\varphi}^K$. We take the faster answer.

5. INTEGRATION INTO TOOLS

Tools typically explore program paths iteratively. At each iteration there is a set of program locations $\{v_1, \ldots, v_k\}$, from which the symbolic execution may continue further. At the beginning, the set contains only program entry location. In each iteration of the symbolic execution the set is updated such that actions of program edges going out from *some* locations v_i are symbolically executed. Different tools use different systematic and heuristic strategies for selecting locations v_i to be processed in the current iteration. It is also important to note that for each v_i there is available an actual path condition φ_i capturing already taken symbolic execution from the entry location up to v_i.

When a tool detects difficulties to cover a particular program location, then using $\hat{\varphi}$ it can restrict selection from the whole set $\{v_1, \ldots, v_k\}$ to only those locations v_i, for which a formula $\varphi_i \wedge \hat{\varphi}$ is satisfiable. In other words, if for some v_i the formula $\varphi_i \wedge \hat{\varphi}$ is *not* satisfiable, then we are guaranteed there is no real path from v_i to the target location. And therefore, v_i can be safely removed from the consideration.

Tools like SAGE, PEX, or CUTE combine symbolic execution with concrete one. Let us assume that a location v_i, for which the formula $\varphi_i \wedge \hat{\varphi}$ is satisfiable, was selected in a current iteration. These tools require a concrete input to the program to proceed further from v_i. Such an input can directly be extracted from any model of the formula $\varphi_i \wedge \hat{\varphi}$.

6. EXPERIMENTAL RESULTS

We have implemented the algorithm (employing the mentioned 'heavyweight' version of Algorithm 2 and supporting mutable arrays, see [27]) in an experimental tool called APC. We also prepared a small set of benchmark programs mostly taken from other papers. In each benchmark we marked a single location as the target one. All the benchmarks have a huge number of real program paths, generated by loops. So, it is difficult to reach the target.

We run PEX and APC on the benchmarks and we measured times till the target locations were reached. This measurement is unfair from PEX perspective, since its task is to cover an analysed benchmark by tests and not to reach a single particular location in it. Therefore, we clarify the right meaning of the measurement. First of all, the measurement is definitely *not* supposed to compare the tools. As we said, it would be unfair. Instead, the only purpose of the measurement is to show, that our algorithm produces *useful* necessary conditions. We show that with a support of PEX as

Table 2: Running times of PEX and APC on benchmarks. Each number in the table represents a running time in seconds. The words SAT, UNSAT and UNKNOWN identify results from queries to Z3 SMT solver with obvious meaning. The marks T/O and M/O represent exceeding of one hour timeout and out of memory error respectively. The column 'Build $\hat{\varphi}$' shows times required for APC to build $\hat{\varphi}$ for benchmarks. The column 'SMT $\hat{\varphi}$' depicts running times and related results from satisfiability queries to Z3 SMT solver for formulae $\hat{\varphi}$. The column 'Build $\hat{\varphi}^{25}$ + SMT $\hat{\varphi}^{25}$' has three data per benchmark. The first number (from the left) identifies time consumed by transformation of $\hat{\varphi}$ to $\hat{\varphi}^{25}$. The second number represents time required by Z3 SMT solver to decide satisfiability of $\hat{\varphi}^{25}$. And the last data represents result from the query to Z3 about satisfiability of $\hat{\varphi}^{25}$. Each number in the column Total of the tool APC is the sum of the following numbers from the same row: The number from the column 'Build $\hat{\varphi}$' and number(s) in bold. Note that numbers in bold identify shorter running times of two parallel computations, whose results are depicted in the last two columns of the table. We discussed the proper setting of Z3 SMT solver with the developers. So, we set the option `produce-models` on for both $\hat{\varphi}$ and $\hat{\varphi}^{25}$, and we set the options `mbqi` and `pull-nested-quantifiers` on only for queries with $\hat{\varphi}$.

	PEX	APC					
Benchmark	**Total**	**Total**	Build $\hat{\varphi}$	SMT $\hat{\varphi}$		Build $\hat{\varphi}^{25}$ + SMT $\hat{\varphi}^{25}$	
Hello	5.257	0.181	0.021	**0.160**	SAT	0.290 + 0.060	SAT
HW	25.05	0.941	0.073	13.84	SAT	**0.698 + 0.170**	SAT
HWM	T/O	4.660	1.715	M/O	–	**2.135 + 0.810**	SAT
MatrIR	95.00	0.035	0.015	**0.020**	SAT	0.491 + 70.80	SAT
WinDriver	28.39	0.627	0.178	4.860	UNKNOWN	**0.369 + 0.080**	SAT
OneLoop	134.0	0.003	0.001	0.010	UNSAT	**0.001 + 0.001**	UNSAT
TwoLoops	64.00	0.003	0.002	**0.001**	UNSAT	0.004 + 0.010	UNSAT

follows. Typical scenario when running PEX on a benchmark is that majority of the code is covered in few seconds (typically up to three). Then PEX needs a longer time to cover the target location or decide that the location is unreachable. We want to show that this longer searching time can be shortened by computing and using a necessary condition $\hat{\varphi}$ for the target location of the benchmark.

It should be easy to modify PEX to detect that it is repeatedly failing for a longer time to cover some program location (like our target). In such cases, the modified PEX could run our algorithm to compute a necessary condition $\hat{\varphi}$ for that location and use it for boosting the subsequent search, for example in the way suggested in the previous section. For all our benchmarks we manually checked that necessary conditions computed by our algorithm are also sufficient ones. Therefore, the subsequent search boosted by $\hat{\varphi}$ mentioned above reduces to a single query to SMT solver producing an input to the benchmark which drives benchmark's execution directly to the target location.

Before we present the results, we discuss the benchmarks (C# listing of all the benchmarks can be found in [27]). Benchmark HWM taken from [1] checks whether an input string contains four substrings: `Hello`, `World`, `At`, and `Microsoft!`. It does not matter at which position and in which order the words occur in the string. The target location can be reached only when all the words are presented in the string. The benchmark consists of four loops in a sequence, where each loop searches for one of the four subwords. Each loop traverses the input string from the beginning and at each position it runs a nested loop checking whether the subword starts at this position. Benchmark HWM is the most complicated one from our set of benchmarks. We also took its two lightened versions presented in [21]: Benchmark HW searches an input string only for subwords `Hello` and `World`, while benchmark Hello searches only for the first one. It is interesting to observe performance of PEX and APC on these three benchmarks, as complexity grows exponentially from Hello to HWM.

Benchmark MatrIR scans upper triangle of an input matrix. The target location is reached if the matrix is bigger than 20×20 and it contains a line with more than 15 scanned elements between 10 and 100.

Benchmarks OneLoop and TwoLoops originate from [21]. They are designed such that their target locations are not reachable. Both benchmarks contain a loop iterated `n`–times. In each iteration, the variable `i` (initially set to 0) is increased by 4. The target location is then guarded by an assertion `i==15` in OneLoop and by a loop `while (i != j + 7) j += 2` in TwoLoops (`j` is initialised to 0 before the loop).

The last benchmark WinDriver comes from a practice and we took it from [12]. It is a part of a Windows driver processing a stream of network packets. It reads an input stream and decomposes it into a two dimensional array of packets. A position in the array where the data from the stream are copied into are encoded in the input stream itself. We marked the target location as a failure branch of a consistency check of the filled in array. It was discussed in the paper [12] that the consistency check can be broken.

The experimental results are depicted in Table 2. They show running times of PEX and APC on the benchmarks. We did all the measurements on a single common desktop computer[1]. For PEX we provide total running times and for APC we in addition provide time profiles of different parts of the computation. As we explained in Section 4, the construction and satisfiability checking of $\hat{\varphi}^K$ runs in parallel with satisfiability checking of $\hat{\varphi}$. Therefore, we take the minimum of the times to compute the total running time of APC. The faster variant is written in boldface in the table. Note that we used the same number $K = 25$ for all the benchmarks to compute $\hat{\varphi}^{25}$ from $\hat{\varphi}$. There is nothing special about this number. It is not related to any of the benchmarks. Actually, all the benchmarks can use different smaller numbers for K to achieve even the same precision as with the chosen one (for example $K = 5$ is sufficient for the benchmark `Hello`).

[1]Intel® Core™ i7 CPU 920 @ 2x2.67GHz, 6GB RAM, Windows 7 Professional 64-bit, MS PEX 0.92.50603.1, MS Moles 1.0.0.0, MS Visual Studio 2008, MS .NET Framework v3.5 SP1, MS Z3 SMT solver v3.2, and boost v1.42.0.

7. RELATED WORK

Early work on symbolic execution [19, 18] showed its effectiveness in test generation. King [19] further showed that symbolic execution can bring more automation into Floyd's inductive proving method [6]. Nevertheless, loops as the source of the path explosion problem were not in the center of interest.

More recent approaches dealt mostly with limitations of SMT solvers and the environment problem by combining the symbolic execution with the concrete one [9, 1, 26, 10, 7, 11, 8, 28, 11, 22]. Although practical usability of the symbolic execution has improved, these approaches still suffer from the path explosion problem. An interesting idea is to combine the symbolic execution with a complementary technique [14, 17, 2, 20, 15]. Complementary techniques typically perform differently on different parts of the analysed program. Therefore, an information exchange between the techniques leads to a mutual improvement of their performance. There are also techniques based on saving of already observed program behaviour and early terminating those executions, whose further progress will not explore a new one [3, 5, 4]. Compositional approaches are typically based on computation of function summaries [7, 1]. A function summary often consists of pre and post condition. Preconditions identify paths through the function and postconditions capture effects of the function along those paths. Reusing these summaries at call sites typically leads to an interesting performance improvement. Moreover, the summaries may insert additional symbolic values into the path condition which causes another improvement. And there are also techniques partitioning program paths into separate classes according to similarities in program states [23, 24]. Values of output variables of a program or function are typically considered as a partitioning criteria. A search strategy Fitnex [29] implemented in PEX [28] guides a path exploration to a particular target location using a fitness function. The function measures how close an already discovered feasible path is to the target.

Although the techniques above showed performance improvements when dealing with the path explosion problem, they do not focus directly on loops. The LESE [25] approach introduces symbolic variables for the number of times each loop was executed. These variables are related to our path counters, but the path counters provide finer information as they are associated to iterations via individual paths through a loop. LESE links the symbolic variables with features of a known grammar generating inputs. Using these links, the grammar can control the numbers of loop iterations performed on a generated input. A technique presented in [13] analyses loops on-the-fly, i.e. during simultaneous concrete and symbolic execution of a program for a concrete input. The loop analysis infers inductive variables, i.e. variables that are modified by a constant value in each loop iteration. These variables are used to build loop summaries expressed in a form of pre a post conditions. Our approach provides more precise analysis of loops than [25, 13], since we introduce an individual path counter for each backbone in a loop (i.e. not a single counter for whole the loop). Therefore, while we are able to precisely express a value of the variable `j` of the left program listed in Section 3.6, the approaches [25, 13] are not able to do so.

In [16] the authors compute a worst-case complexity bound on the number of visits to a given program location l for any execution of the program. Heart of the computation is an analysis of a part of the program consisting of paths going from the location l back to it. The program part is first transformed into formula T using a simple data-flow analysis, and then there is computed a transitive closure T' of T based on convex-theory-like assumption. If the program part contains a loop, then the described process is recursively repeated, where the location l now represents a single entry location to the loop. The computed closure T' (viewed as a loop summary in this case) is then inserted just before l and back-edges of the loop are discarded. The resulting closure T' of whole the program part is then used to compute ranking functions. These functions are inferred using pattern-matching technique and are used to express the worst-case complexity bound. Values of the variables in the bound are linked to the program input using a backward symbolic execution from the analysed part towards program entry. Our approach differs in several key aspects: First, our analysis is applied to all program loops, instead of analysing only paths going from l back to it. Second, we use pattern-matching to compute precise values of variables rather then computing ranking functions. And finally, we use path counters to compute iterated symbolic memory and looping condition (i.e. the summary), while the summary in [16] is the closure of the formula $T(\vec{x}, \vec{x}')$ relating values of live variables $\vec{x}$ at the location l with values $\vec{x}'$ of these variables in the subsequent visit of l.

As far as we know, the algorithm presented in [21] is the most related to our work. It also tackles the reachability of a given target location, it focuses on program loops, and it is also supposed to be used as a heuristic for test generation tools. Nevertheless, there are several conceptual differences between both algorithms. The algorithm in [21] first transforms analysed program into tree-like structure of chains. A chain is basically a linear sequence of instructions along a cyclic path in the analysed program. The remaining analysis is performed on the chains. For each chain with sub-chains the algorithm infers a constraint system. Then the algorithm applies symbolic execution on chains. When a symbolic execution reaches an instruction of a chain where sub-chains are attached, the constraint system (of the chain) is used as an oracle to decide, where to continue next. It is either possible to step into some of the sub-chains or to continue further along the chain. The target location is reached when the last instruction of the root chain is executed. Besides differences in approaches of both algorithms, there are also differences in their integration into symbolic execution tools. On one hand, our algorithm applies fast and finite program analysis producing a necessary condition. A tool then applies its own symbolic exploration of program paths boosted with the necessary condition. On the other hand, the algorithm in [21] applies its own symbolic execution on chains. This execution can be long or even infinite. Therefore, a tool have to continue its general search until [21] gives a result.

8. CONCLUSION

We presented a symbolic-execution-based algorithm computing, for a given program and a target location in it, a nontrivial necessary condition on input symbols for reaching of the target location. In particular, if an input to the program is not a model of the condition, then the execution of the program on that input will definitely miss the target location. The key part of the algorithm is computation of loop summaries. A loop summary consists of iterated symbolic memory representing overall effect of all changes to program state done by all executions of the loop, and looping condition, which is a non-trivial formula implied by all path conditions of all symbolic executions of the loop. We further proposed a use of the algorithm in test-generation tools based on symbolic execution, and we showed that the integration is straightforward. We also implemented our algorithm in an experimental tool and we empirically checked, for a small set of benchmarks, that PEX could cover the benchmarks faster with our algorithm.

9. ACKNOWLEDGMENTS

The authors are supported by the Czech Science Foundation, grant No. P202/12/G061.

10. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of TACAS*, pages 367–381. Springer, 2008.

[2] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from tests. In *Proceedings of ISSTA*, pages 3–14. ACM, 2008.

[3] P. Boonstoppel, C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of TACAS*, pages 351–366. Springer-Verlag, 2008.

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

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of CCS*, pages 322–335. ACM, 2006.

[6] R. W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967.

[7] P. Godefroid. Compositional dynamic test generation. In *Proceedings of POPL*, pages 47–54. ACM, 2007.

[8] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of PLDI*, pages 206–215. ACM, 2008.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of PLDI*, pages 213–223. ACM, 2005.

[10] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *Proceedings of EMSOFT*, pages 207–216. ACM, 2008.

[11] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of NDSS*, pages 151–166, 2008.

[12] P. Godefroid, X. Levin, and X. Elkarablieh. Precise pointer reasoning for dynamic test generation. In *Proceedings of ISSTA*, pages 129–140. ACM, 2009.

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

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

[15] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proceedings of SIGSOFT*, pages 117–127. ACM, 2006.

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

[17] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *Proceedings of TACAS*, pages 262–276. Springer-Verlag, 2009.

[18] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Software Eng.*, 3:266–278, 1977.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[20] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *Proceedings of TACAS*, pages 178–181. Springer-Verlag, 2009.

[21] J. Obdržálek and M. Trtík. Efficient loop navigation for symbolic execution. In *Proceedings of ATVA*, pages 453–462. LNCS, 2011.

[22] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of ISSTA*, pages 34–44. ACM, 2011.

[23] D. Qi, H. D. T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *Proceedings of ESEC/FSE*, pages 278–288. ACM, 2011.

[24] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multipe-path symbolic execution. In *Proceedings of ISSTA*, pages 195–206. ACM, 2010.

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

[26] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of ESEC*, pages 263–272. ACM, 2005.

[27] J. Strejček and M. Trtík. Abstracting path conditions. Technical report, 2011. `http://www.fi.muni.cz/~xtrtik2/apc.pdf`.

[28] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proceedings of TAP*, pages 134–153. Springer, 2008.

[29] X. Tao Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of DSN*, pages 359–368. IEEE, 2009.