

Compact Symbolic Execution

Marek Trtík

Faculty of Informatics, Masaryk University, Brno, Czech Republic
 trtik@fi.muni.cz

Abstract. We present a generalisation of King’s symbolic execution technique called compact symbolic execution. It is based on a concept of templates: a template is a declarative parametric description of such a program part, generating paths in symbolic execution tree with regularities in program states along them. Typical sources of these paths are program loops and recursive calls. Using the templates we fold the corresponding paths into single vertices and therefore considerably reduce size of the tree without loss of any information. There are even programs for which compact symbolic execution trees are finite even though the classic symbolic execution trees are infinite.

1 Introduction

Classic symbolic execution as proposed by King in 1976 [8] systematically explores all real paths in an analysed program. There is typically huge (or even infinite) number of real paths even for very small and simple programs. Therefore, exploration of the real paths becomes a serious problem. We speak about the *path explosion problem*.

Compact symbolic execution also explores all real program paths, but in a very compact manner. We analyse a given program *before* we start its symbolic execution. We look for those parts of the program, which might produce real paths with some regularities in program states along them. Typically, program loops and recursion produces these regularities. We analyse the program parts independently from the remainder of the program. If the analysis of a part succeeds, then a result is a *template*, i.e. a declarative parametric description of the complete behaviour of the analysed part. Therefore, an output from the program analysis is a set of templates. Now we can execute the program symbolically with the templates. Until we reach some of the successfully analysed program parts, we proceed just like in classic symbolic execution. Let us now suppose we have just reached such a part. Having a template for the part, we do not need to symbolically execute interior of the part. We just instantiate the template into the end of the current path and then we *jump* behind the part, where we continue with classic symbolic execution again.

Let us consider a symbolic execution reaching a loop. The execution may fork into a huge number of other symbolic executions during the execution of the loop. Each such execution has its own path in symbolic execution tree of classic symbolic execution. But having a template for the loop, we represent

all these paths by a single one with the instantiated template. In other words, a single path explored by compact symbolic execution may represent a huge number of paths explored by classic symbolic execution. And that is the cause of the considerable space savings of compact symbolic execution. On the other hand, we will see that compact symbolic execution has higher requirements to performance of SMT solvers than classic one.

The worst case for compact symbolic execution is, when we fail to compute any template for a given program. Compact symbolic execution then reduces to classic one, and we gain no space savings.

2 Overview

In this section we give an intuition of compact symbolic execution. For simplicity of presentation we use the following definition of a program. Although our programs are simple they support typical imperative constructs and recursion.

Definition 1 (Program) *A program is a collection of functions and global variables. Each function has its own local variables. All program variables and functions have different names. Exactly one function is marked as starting one. Each function is represented as an oriented graph. Vertices in the graph identify program locations, while edges define transitions between them. We distinguish a single entry and exit location in each graph. There is no in-edge to entry location and there is no out-edge from the exit one. We label edges by actions to be taken when moving between connected locations. An action can be*

- (1) *An assignment of the form $\langle \text{variable} \rangle := \langle \text{expression} \rangle$,*
- (2) *Call by value statements*
 - (a) $\langle \text{variable} \rangle := \langle \text{function-name} \rangle (\langle \text{arg-list} \rangle)$, or
 - (b) $\langle \text{function-name} \rangle (\langle \text{arg-list} \rangle)$
- (3) *A return value statement **ret** $\langle \text{expression} \rangle$,*
- (4) ***skip** statement, which does nothing, or*
- (5) *A boolean expression over program variables.*

*If an edge $e = (u, v)$ is labelled by one of the actions (1)-(4), then out-degree of u is 1. Otherwise, label of e is an action (5), out-degree of u is 2 and its out-edges are labelled by boolean expressions γ and $\neg\gamma$. No action (2) can reference the starting function and no entry nor exit location is incident with an edge having an action (2). Each function f is assigned a unique global variable **ret** _{f} used for actions (2a) to save a return value being later assigned to the destination variable. And for simplicity we do not consider pointer arithmetic nor heap allocations. We prevent invalid operations in actions (like division by zero, etc.) by branchings into error locations. An error location is any location with a single out-edge heading back to that location and it is labelled with **skip** action.*

We can see an example of a program at Figure 1 (a). The depicted function `linSrch` returns the least index i into the array A such that $A[i] == x$. If x is not in A at all, then it returns -1 .

We first briefly describe classic symbolic execution as proposed by King [8]. Instead of passing concrete data into parameters of the starting function, we pass symbols from a set $\{\alpha_0, \alpha_1, \dots\}$. Let us suppose we pass symbols α_0 and α_1 to variables **a** and **b** respectively. After executing an action $c := 2 * a + b$ the variable **c** will contain a *symbolic expression* $2\alpha_0 + \alpha_1$ as its value. *Symbolic memory* is a function θ from program variables to a set of symbolic expressions. We further maintain a boolean symbolic expression φ called *path condition*. It represents a complete identifier of a particular program path taken during an execution. φ is initially *true* and it can be updated at program branchings. Let θ be a symbolic memory having $\theta(a) = \alpha_0$, $\theta(b) = \alpha_1$ and $\theta(c) = 2\alpha_0 + \alpha_1$ and let $c - a > 2 * b$ and $c - a \leq 2 * b$ be actions of out-edges of an branching location. For the first action we proceed as follows. We evaluate the action in θ . The result is a boolean symbolic expression $\alpha_0 + \alpha_1 > 2\alpha_1$. If $\varphi \rightarrow (\alpha_0 + \alpha_1 > 2\alpha_1)$ is satisfiable, we update φ to $\varphi \wedge (\alpha_0 + \alpha_1 > 2\alpha_1)$ and we continue the execution by crossing the edge having the action. Then we proceed similarly for the second action. Note that if both implications are satisfiable, we fork the execution into two parallel and independent executions. Besides a symbolic memory and a path condition we commonly have a call stack Ξ and we also need to identify a current program location l . Putting all the things together we get a *program state* represented by a tuple $s = (\theta, \varphi, \Xi, l)$. Note that we understand a call stack record as pairs (σ, l) , where l is a return location and σ is a restriction of a symbolic memory to local variables. Further, we commonly describe the symbolic execution of a program by a tree structure called *symbolic execution tree*. Vertices of the tree are related to program locations visited during the execution and edges reflect transitions between the locations. Each vertex of the tree is labelled by a related program state. But instead of labels *T* and *F* for branching edges (as proposed by King), we label them by evaluated actions of the branching edges. Figure 1 (b) depicts a part of symbolic execution tree of the example program from Figure 1 (a) (with omitted program states labelling the vertices). Please ignore grey regions in the tree for now. We assume that classic symbolic execution of the program started with an initial symbolic memory $\theta = \{(i, \alpha_0), (n, \alpha_1), (x, \alpha_2), (A, \alpha_3)\}$.

We often use the following dot-notation to access elements of tuples. If $s = (\theta, \varphi, \Xi, l)$ is a program state, then $s.\theta$ denotes its symbolic memory, $s.\varphi$ denotes its path condition, $s.\Xi$ is its call stack and $s.l$ is a current program location. Further, if u is a vertex of symbolic execution tree, then $u.s$ denotes program state labelling the vertex. And instead of $u.s.\theta$, $u.s.\varphi$, $u.s.\Xi$ and $u.s.l$ we simply write $u.\theta$, $u.\varphi$, $u.\Xi$ and $u.l$. Finally, if Ξ is a call stack then we use dot-notation to access record at the top of the call stack. So, for example $\Xi.l$ denotes return location of record at the top of Ξ .

Symbols $\{\alpha_0, \alpha_1, \dots\}$ in classic symbolic execution represent input values to whole program. We generalise this concept to allow independent symbolic execution of *parts* of an analysed program independently to the remainder. Each such a part uses the symbols $\{\alpha_0, \alpha_1, \dots\}$ relative to a chosen entry location to the part. Then using a *composition* of program states (defined later) we can express any run of classic symbolic execution as a composition of program states

resulting from analyses of the parts. Let $s = (\theta, \varphi, \Xi, l)$ be a program state resulting from a symbolic execution from a program location l_0 (e.g. the entry location of the starting function), up to an entry location l of an independently analysed program part. Let $s' = (\theta', \varphi', \Xi', l')$ be a program state resulting from the analysis of the part, i.e. s' represents a symbolic execution from the entry location l to some exit location l' from the part. Then $s \circ s' = (\theta \circ \theta', \varphi \wedge \theta(\varphi'), \Xi \circ (\theta \circ \Xi'), l')$ is composed program state representing symbolic execution from l_0 to l' through the analysed part (entered in location l). We can see that composition of program states is implemented as composition of their individual components. We discuss very details of these operations in Section 3. Only note that composed path condition is $\varphi \wedge \theta(\varphi')$ rather than $\varphi \wedge \varphi'$. This is because φ' may contain some symbols. But they are related to the entry location l of the analysed part and not to the location l_0 . Therefore, we have to compose φ' with θ first to express φ' in terms of symbols relative to location l_0 . We do the similar effect of shifting symbols from location l to l_0 in the compositions $\theta \circ \theta'$ and $\theta \circ \Xi'$.

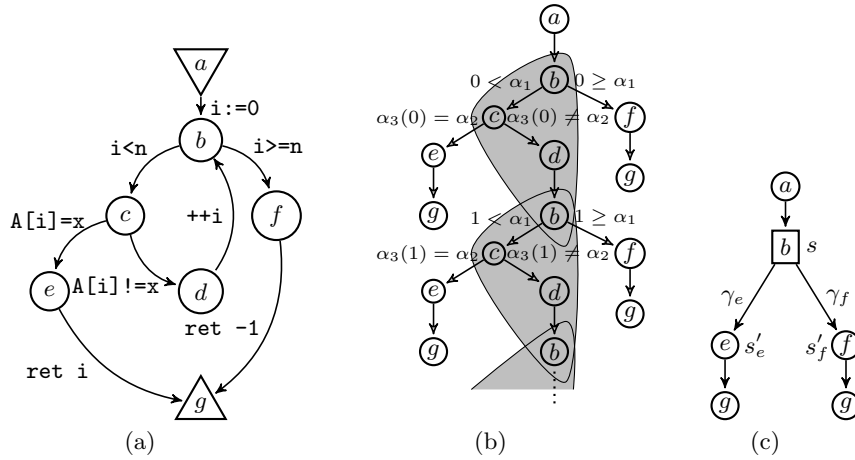


Fig. 1. (a) A program with a function `linSrch(A, n, x)`. (b) Symbolic execution tree of function `linSrch`. (c) Compact symbolic execution tree of function `linSrch`.

In symbolic execution tree at Figure 1 (b) there is a single path highlighted by a sequence of grey regions. Vertices in each region are related to the same sequence of program locations: b, c, d, b . Moreover, we enter the path at vertex referencing location b and we can leave the path either by stepping into a vertex referencing location e or into a vertex referencing location f . Let us denote the entry vertex into the path as b_0 and the exit vertices from the path referencing locations e and f as e_0, e_1, \dots and f_0, f_1, \dots respectively being indexed from the top down. Our goal is to completely eliminate the path in grey from the tree, while still representing all real program paths. One way to do so is to represent

whole the path by a single vertex, b say, with two direct successors. The first successor, e say, represents all the exit vertices e_i from the path and the second, f say, representing all the exits f_i . Note that names of the vertices b , e and f also represent program locations they reference. We label the vertex b by the program state labelling b_0 . But the question is what program states we should assign to the vertices e and f . Note that two different vertices e_i and e_j may be labelled by different program states. So, for the vertex e we need to introduce a program state $e.s[\kappa]$, parametrised by a *parameter* κ , such that each program state $e_i.s$ can be equivalently expressed by $e.s[\kappa]$, when κ is substituted by some number ν . Of course, for different states $e_i.s$ and $e_j.s$ there are different numbers, say ν_i and ν_j , for parameter substitution. We similarly need a parametrised program state $f.s[\kappa]$ for the vertex f . We compute the states $e.s[\kappa]$ and $f.s[\kappa]$ before we start symbolic execution of the program from the Figure 1 (a) by analysing the following its part. The part consists of all the locations b, c, d, e, f discussed above and of all the edges between them. Note that the sequence b, c, d, b of locations forms a cyclic path inside the analysed part. This cycle is actually the source of the path in grey regions. Nevertheless, we want to describe program states at *exits* from the part. The exits from the part are target vertices of those edges of the part, which do not belong to the cycle. Therefore, locations e and f are the exits from the part. We also identify the location b as entry location into the part, since we can enter the part by stepping into location b . The part is completely defined now. We analyse it independently from the remainder of the program. It mainly means that if we use some symbols α_i in the analysis, then they are related to the entry location b of the part and not to the entry location of the whole program. At this point we are more concerned about formulation of a result from the analysis and its usage then the analysis itself. Therefore, we postpone its description to Section 5. We assume here that key properties $e.s[\kappa]$ and $f.s[\kappa]$ from the analysis are already computed, so we may formulate an output from the analysis of the part as the following *template*

$$t = (b, 2, \{(\theta_e, \varphi_e, [], e)[\kappa], (\theta_f, \varphi_f, [], f)[\kappa]\}),$$

where b is the entry location to the analysed part, the number 2 identifies number of following parametrised program states and the remaining two tuples are the parametrised program states $e.s[\kappa]$ and $f.s[\kappa]$ respectively. Note that $[]$ identifies empty call stack. The template contains all the information we need to build compact symbolic execution tree, where the path in grey is folded as described above.

Let us symbolically execute the program at Figure 1 (a) with the template t . We construct a compact symbolic execution tree during the execution. The tree is depicted at Figure 1 (c). We apply classic symbolic execution, until we reach the entry location $t.b$. Let b be the vertex in the tree, when we reach the location $t.b$ and let s be the program state $b.s$. We now instantiate the template. Since we have exactly two program states in t , we create exactly two successor vertices e and f of the vertex b in the tree. The vertices e and f references locations $t.e$ and $t.f$ respectively and they are further labelled by program states

$s \circ (t.\theta_e, t.\varphi_e, [], t.e)[\kappa]$ and $s \circ (t.\theta_f, t.\varphi_f, [], t.f)[\kappa]$ respectively. We finish the instantiation of t by creating edges (b, e) and (b, f) labelled by symbolic expressions $s.\theta\langle t.\varphi_e[\kappa] \rangle$ and $s.\theta\langle t.\varphi_f[\kappa] \rangle$ respectively. The situation is also depicted at Figure 1 (c). Then we continue from both vertices e and f independently using classic symbolic execution again. These both executions reaches function exit location g in one step and compact symbolic execution terminates.

Let us now have a look at Figure 2 (a) depicting a program with a function `countIf`. The function counts number of elements in array `A` having values equal to `x`. We show the symbolic execution tree of the program at Figure 2 (b). There we can see several sequences of grey regions. According to our experience with the previous example we can easily detect that all that paths in grey are generated by a single program part consisting of locations c, d, e, f, g and edges between them. But there are two cyclic paths $\pi = c, d, e, f, c$ and $\pi' = c, d, f, c$ inside the part. Nevertheless, the grey regions highlight only the cycle π . So, we ignore the cycle π' and π is therefore the only cycle we consider. The remainder is now obvious. The locations f and g are exits from the part and c is the entry location into the part. The analysis of the path (discussed later in Section 5) computes the following template

$$t = (c, 2, \{(\theta_f, \varphi_f, [], f)[\kappa], (\theta_g, \varphi_g, [], g)[\kappa]\})$$

Compact symbolic execution with the template t computes compact symbolic execution tree depicted at Figure 2 (c). The tree is basically a single link list. Note that we instantiate the template each time we reach the location c . But for each such instantiation we need a fresh parameter to prevent parameter collisions from previous instantiations. We assume we have infinitely many different names for the parameters. Therefore, expressions and program states at Figure 2 (c) are as follows: $\gamma_g^i = s_c^i.\theta\langle t.\varphi_g[\kappa_i] \rangle$, $\gamma_f^i = s_c^i.\theta\langle t.\varphi_f[\kappa_i] \rangle$, $s_g^i = s_c^i.\theta \circ (t.\theta_g, t.\varphi_g, [], g)[\kappa_i]$ and $s_f^i = s_c^i.\theta \circ (t.\theta_f, t.\varphi_f, [], f)[\kappa_i]$.

The sequences of grey regions in the tree at Figure 2 (b) goes bottom left. But imagine they would go bottom right. Then each region would represent a sequence of program locations c, d, f, c . If we analysed closer these sequences of grey regions, we would realise that there is a part of the program from Figure 2 (a) consisting of vertices c, d, f, e, g , where c, d, f, c is the only cycle in the part, c is the entry location into the part and locations e and g are exits from the part. If we further built a template from the part and run compact symbolic execution with it, we would also receive a compact symbolic execution tree forming basically a single linked list.

Besides cyclic paths, recursive calls also produce real program paths with regularities in program states along them. At Figure 3 (a) there is a recursive function `linSrchRec` which is equivalent to the function `linSrch` discussed before. Symbolic execution tree of the function is depicted at Figure 3 (b). The root of the tree is the left-most vertex referencing program location a . There are two sequences of grey regions. The top sequence represents recursive calls, while the bottom sequence represents returning from the calls. We see that top sequence goes from left to the right. The bottom sequence goes in the opposite

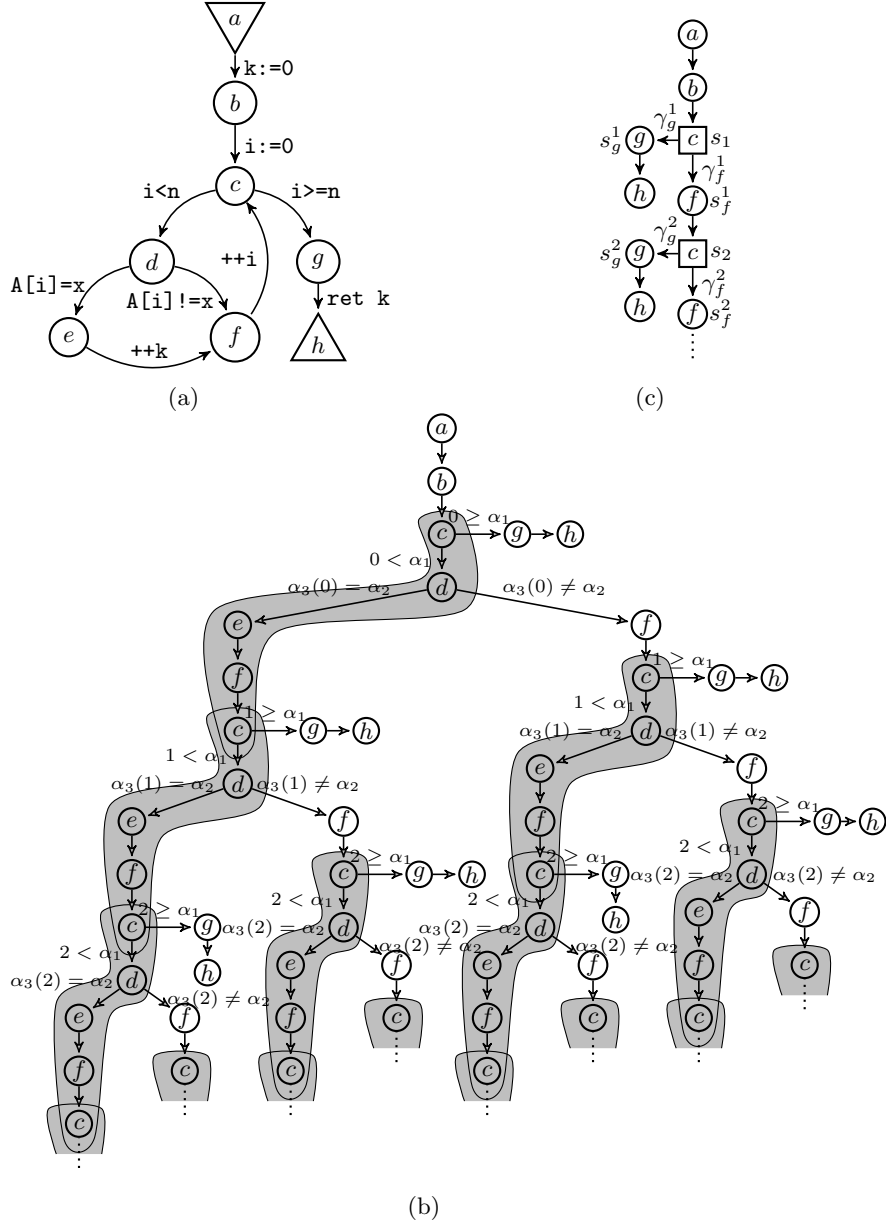


Fig. 2. (a) A program with a function `countIf(A,n,x)`. (b) Symbolic execution tree of function `countIf`. (c) Compact symbolic execution tree of function `countIf`.

direction. We can further see there is one to one correspondence between regions of both sequences. Below each region in the top sequence, there is a single region of bottom sequence. Paths in both sequences of regions are connected in the

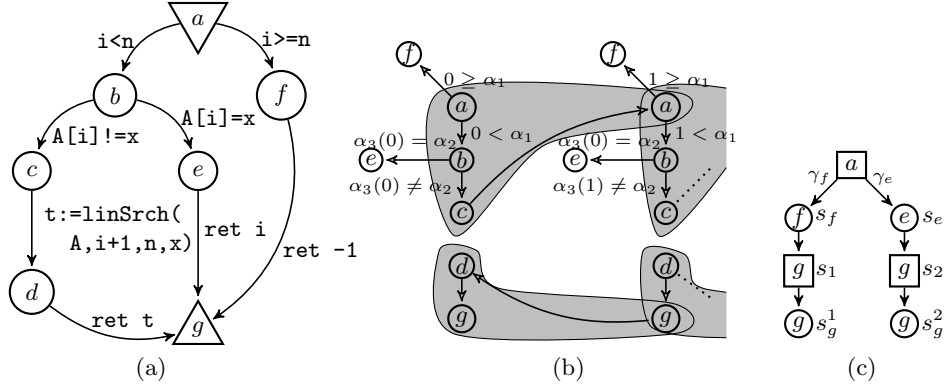


Fig. 3. (a) A program with a recursive function `linSrch(A, i, n, x)`. (b) Symbolic execution tree of the recursive function `linSrch`. (c) Compact symbolic execution tree of the recursive function `linSrch`.

tree. But this is not shown in the figure. The connection happens, when all the recursive calls are done and some basic case is executed in the recursive function. Then we get to the path of the bottom regions.

Let us first focus on the path at the top sequence of regions. Vertices in each region are related to the same sequence of program locations: a, b, c, a . Moreover, we enter the path in a vertex referencing location a and we can leave the path either by stepping into a vertex referencing location f or into a vertex referencing location e . If we look at the program (at Figure 3 (a)), the sequence a, b, c, a forms a cyclic path in it. Of course, the edge (c, a) is not explicit in the program. But we consider it as a meta-edge labelled by an action simulating the effect of the function call, as defined by action of edge (c, d) . We now define a program part, say P_1 , consisting of the cyclic path, the entry location a and two exit locations f and e . The part represent the phase of recursive calls of the function `linSrch`.

Now we similarly analyse the path in bottom sequence of regions. Each region repeats the same sequence of program locations g, d, g . The path is entered in vertex referencing location g , but there is no exit from the path. The sequence g, d, g of locations forms a cyclic path in the program (at Figure 3 (a)). Note, that we assume there is an artificial edge (g, d) enclosing the cycle. Action of this edge is supposed to simulate the effect of return from the function call, as defined by action of edge (c, d) . We want to define a program part P_2 representing the phase of returning from recursive calls. We have the cyclic path and we have the entry location g to the part. But there is no exit from the part. Obviously, the recursive calls ends in location g , where we leave the function. Therefore, our exit location is g and we have the program part P_2 . Note that if we want to formally match the exit location detection algorithm introduced for previous examples, we may imagine there is an edge from g back to g and labelled by `skip` action.

For the program parts P_1 and P_2 we compute the following templates t_1 and t_2 as described in the previous examples.

$$\begin{aligned} t_1 &= (a, 2, \{(\theta_f, \varphi_f, [], f)\llbracket \kappa \rrbracket, (\theta_e, \varphi_e, [], e)\llbracket \kappa \rrbracket\}) \\ t_2 &= (g, 1, \{(\theta_g\llbracket \kappa \rrbracket, true, [], g)\}). \end{aligned}$$

Note that the path condition of t_2 is simply *true*, since we cannot escape from the path. In other words, as there is no branching along the path, the path condition cannot be updated from its initial value *true*. It is important to note, that both templates use exactly the same parameter. The use of the same parameter creates a link between the number of recursive calls and number of returns from them. Having the templates we are able to formulate the template t for the recursive function `linSrchRec`.

$$t = (a, 2, \{(\theta_f, \varphi_f, [], f)\llbracket \kappa \rrbracket, (\theta_e, \varphi_e, [], e)\llbracket \kappa \rrbracket\}, \theta_g\llbracket \kappa \rrbracket, g)$$

The template t contains whole the template t_1 , but it took only symbolic memory θ_g and the exit location g from the template t_2 .

We are ready to start compact symbolic execution with the template t . Symbolic execution tree for the program is depicted at Figure 3 (c). First we step into the program location a . The tree contains only the root vertex referencing location a . The location a is the entry location of t . Hence, we instantiate the first part of t (related to phase of recursive calls, i.e. related to t_1) into the tree. The number 2 in t identifies, that the root will have two successor vertices referencing locations f and e and they will be labelled by program states $s_f = (\theta_f, \varphi_f, [(t, \kappa)] \circ [], f)\llbracket \kappa \rrbracket$ and $s_e = (\theta_e, \varphi_e, [(t, \kappa)] \circ [], e)\llbracket \kappa \rrbracket$ respectively. Note that we omitted composition of these states with the initial program state labelling the root. We could do that, since composition of initial program state with any other state produces the other state again. Also note that call stacks of both states (i.e. $[]$) are composed with a call stack containing a single special record of the form (t, κ) . This type of call stack record is introduced only for templates of recursive functions. First of all, this single record represents any number of subsequent recursive calls done by classic symbolic execution. And the record also saves reference to the template t and the parameter κ used in the instantiation. We note that edges from the root to its successors are labelled by expressions $\gamma_f \equiv t.\varphi_f\llbracket \kappa \rrbracket$ and $\gamma_e \equiv t.\varphi_e\llbracket \kappa \rrbracket$. Having computed successors of the root, we continue by classic symbolic execution independently from both of these vertices, until we reach the location g . For both the executions we do the same think at the location g . Let us consider execution continuing from the successor referencing location f . We need to instantiate the second part of the template representing returns from the recursive calls. So, we remove the record (t, κ) from the top of the call stack, but we take the template t and the parameter κ stored in the record (t, κ) . In general, between both instantiation parts of a given template, there might be executed any code, there can be instantiated many other templates and there can even be instantiated the same template several times always with different (fresh) parameters. That is why we save the template and the parameter in the stack record. Let s_1 be a program state of

the current leaf vertex of the tree. We create its only successor vertex labelled with program state $s_g^1 = (s_1.\theta \circ t.\theta[\![\kappa_1]\!], s_1.\varphi, [], g)$. We see that there are two differences between states s_1 and s_g^1 . First of all call stack of s_g^1 does not contain the special record (t, κ) as we have popped it from the stack. And second, the symbolic memory of s_g^1 is the composition $s_1.\theta \circ t.\theta[\![\kappa_1]\!]$. Further classic symbolic execution from the vertex terminates, since we are leaving exit location of the starting function. We proceed similarly for the other run of symbolic execution (from the second successor of the root), where we get the final program state $s_g^2 = (s_2.\theta \circ t.\theta[\![\kappa]\!], s_2.\varphi, [], g)$.

To summarise, a general scheme for compact symbolic execution of the examples above is as follows. We enumerate parts in a given program producing paths with regularities in program states along them. Such sources are mainly cyclic paths and pairs of cyclic paths representing recursion. For each enumerated part we compute a template. Then we run compact symbolic execution with the computed templates.

3 Definition

In this section, we give precise definition of templates parametrised by a single parameter. Templates for recursion consists of two parts instantiated independently into symbolic execution tree. These instances share the same parameter. We therefore show a process of information passing between different instances of the same template. And we further present compact symbolic execution algorithm using templates with one parameter with possible information exchange between instances. We start with basic terms valid for compact symbolic execution with any kinds of templates. We assume for the rest of this section that P is a program.

An injective function Θ from a set of all program variables of P to a set of symbols $\{\alpha_0, \alpha_1, \alpha_2, \dots\}$ is an *initial symbolic memory* of P . For each program variable \mathbf{a} its symbol $\Theta(\mathbf{a})$ represents some yet unknown value of that variable. So, $\Theta(\mathbf{a})$ must belong to a domain of \mathbf{a} (i.e. $\Theta(\mathbf{a})$ is of \mathbf{a} 's type). Further, *numeric symbolic expression* is application of operators to numeric constants and symbols. *Boolean symbolic expression* is either an equality or inequality predicate over numeric symbolic expressions, or an application of logical connectives to other boolean symbolic expressions. *Symbolic expression* is either numeric or boolean symbolic expression. We have already given the definition of symbolic memory, call stack and program state in Section 2. But we in addition define for any program state $s = (\theta, \varphi, \Xi, l)$ that $\theta(\mathbf{a}) = \Theta(\mathbf{a})$, for each local variable \mathbf{a} undefined at location l . Also note that Θ is just a special symbolic memory.

The pseudo-code of Algorithm 1 represents two algorithms. If we consider only unmarked lines, we get algorithm of classic symbolic execution. If we add lines marked with \square we get algorithm of compact symbolic execution with templates with a single parameter. The lines marked by $*$ are responsible for construction of symbolic execution tree. Obviously, both classic and compact sym-

bolic executions can appear at both versions: with and without construction of the tree.

We now describe the algorithm of classic symbolic execution. At line 2, there we create initial program state and then we insert it into a queue Q . The queue Q keeps all program states for which we have not been computing successor program states yet. Until Q becomes empty, we iterate the loop at lines 5–38. At line 7 we detect whether actually processed program state s is final or not. If it is not, we compute its successors at line 32. In short, the function `computeClassicSuccessors` either executes actions of out-edges from location $s.l$ or it resolves return from a call, if $s.l$ is a function exit location. We already gave an intuition how to symbolically execute actions at the beginning of Section 2. We further see at line 34 that we discard all successors of s , whose path conditions are not satisfiable. Discarded states do not represent real behaviour of the program.

Now we focus on $*$ -version of the algorithm. We create root of the tree labelled by the initial program state at line 4. When processing a state s inside the loop we take the only leaf in the tree labelled with s at line 33. We compute its successor vertices at lines 36 and 37. Note that the successors are labelled by successor states of s .

We have to postpone description of \square -version of the algorithm, until we have properly defined templates the algorithm uses. The first step toward the definition is introduction of parameters and their substitution.

We distinguish a set $\{\kappa, \tau, \kappa_1, \tau_1, \kappa_2, \tau_2, \dots\}$ of variables called *parameters*, ranging over non-negative integers. We extend numeric symbolic expressions such that they may also contain application of operators to parameters. We allow boolean symbolic expression to contain quantification of parameters. We further naturally extend symbolic memories, call stacks and program states to contain symbolic expressions with parameters. When we want to emphasise that κ is a set of all parameters appearing in a symbolic expression φ , we denote it as $\varphi[\kappa]$. And if we want to emphasise that a symbolic expression φ does not contain any parameter, we denote it as $\varphi[]$. We naturally extend the notations above for symbolic memories, call stacks and program states.

We now describe substitution of parameters. Each function from a finite set of parameters to non-negative integers is *valuation*. Let $\varphi[\kappa]$, $\theta[\kappa]$, $\Xi[\kappa]$ and $s[\kappa]$ be a symbolic expression, a symbolic memory, a call stack and a program state respectively, $\kappa \neq \emptyset$ and ν be a valuation defined for all parameters in κ . Then we compute $\varphi[\nu]$ from $\varphi[\kappa]$ such that we substitute all parameters in φ by related integers in ν . We compute $\theta[\nu]$ from $\theta[\kappa]$ such that we substitute all parameters in all the expressions in θ by related integers in ν . Substitution of call stack parameter is a bit more complicated, since we introduced the special form (t, κ) of a stack record in the last example of Section 2. Therefore, to prepare ground for stack equivalence, we compute $\Xi[\nu]$ from $\Xi[\kappa]$ in the following two steps: (1) We update each record (σ, l) of the call stack Ξ to $(\sigma[\nu], l)$ (note that σ is basically symbolic memory, only restricted to local variables). (2) Each record of the special form (t, κ) in the call stack from the previous step is either

Algorithm 1: executeSymbolically

Input: P - program to be executed
 d - set of template detectors (only in \square -version)
Output: E - set of final program states
 T - symbolic execution tree of P (only in $*$ -version)

```

□ 1 Let  $p$  be a set of all templates detected in  $P$  by detectors  $d$ 
2  $s_0 := (\Theta, true, \square)$ , entry location of the starting function
3 Let  $Q$  be a queue of program states initially containing only  $s_0$ 
* 4 Create a root vertex of  $T$  labelled with  $s_0$ 
5 repeat
6   Extract the first program state  $s$  from  $Q$ 
7   if  $s.l$  is the exit location of the starting function or an error location then
8     Insert  $s$  into  $E$ 
9   else
10     $S := \emptyset$ 
11    if  $\text{top}(s.\Xi) = (t, \kappa) \wedge s.l = s.\Xi.t.l'$  then /* returning from recursion */
12       $t := s.\Xi.t$ 
13       $\kappa := s.\Xi.\kappa$ 
14      Replace all occurrences of the former parameter in  $t$  by  $\kappa$ 
15       $s' := (s.\theta \circ t.\theta[\kappa], s.\varphi, \text{pop}(s.\Xi), t.l')$ 
16      Insert  $s'$  into  $S$ 
17    else
18       $p' := \text{getTemplatesAt}(s.l, p)$ 
19      if  $p' \neq \emptyset$  then
20         $t := \text{chooseTemplate}(p')$ 
21         $\kappa := \text{getFreshParam}()$ 
22        Replace all occurrences of the former parameter in  $t$  by  $\kappa$ 
23        if  $t$  is a recursion template then /* recursive calling */
24          foreach  $i = 1, \dots, t.n$  do
25             $s' := s \circ (t.\theta_i, t.\varphi_i, [(t, \kappa)] \circ t.\Xi_i, t.l_i)[\kappa]$ 
26            Insert  $s'$  into  $S$ 
27          else /*  $t$  is a general template with one parameter */
28            foreach  $i = 1, \dots, t.n$  do
29               $s' := s \circ (t.\theta_i, t.\varphi_i, t.\Xi_i, t.l_i)[\kappa]$ 
30              Insert  $s'$  into  $S$ 
31          else /* applying classic symbolic execution step */
32             $S := \text{computeClassicSuccessors}(P, s)$ 
* 33 Let  $u$  be a leaf of  $T$  whose label is  $s$ 
34 foreach program state  $s' \in S$  such that  $s'.$  $\varphi$  is satisfiable do
35   Insert  $s'$  at the end of  $Q$ 
* 36 Insert a new vertex  $v$  labeled with  $s'$  into  $T$ 
* 37 Insert an edge  $(u, v)$  into  $T$ 
38 until  $Q$  becomes empty
39 return  $E$ 
* 40  $T$ 
```

discarded, if $\nu(\kappa) = 0$, or it is replaced by $\nu(\kappa)$ records (\perp, \perp) , where symbol \perp represent *any* possible content. Therefore, the record (\perp, \perp) represents any possible stack record (the first \perp in the record represents any possible content of σ and the second \perp represents any possible program location).

We often use the following simplified notation. If an expression φ contains exactly one parameter κ and a $\{(\kappa, \nu)\}$ is a valuation, then we write $\varphi[\![\kappa]\!]$ and $\varphi[\![\nu]\!]$ instead of $\varphi[\![\{\kappa\}]\!]$ and $\varphi[\![\{(\kappa, \nu)\}]\!]$ respectively. The notation also applies to symbolic memories, call stacks and program states.

Next we define composition of program states and equivalence between them. We also express some basic equivalences for compositions.

Definition 2 (Composition) Let $\Xi = [r_0, \dots, r_m]$ and $\Xi' = [r'_0, \dots, r'_n]$ be call stacks and $s = (\theta, \varphi, \Xi, l)$ and $s' = (\theta', \varphi', \Xi', l')$ be program states. Then composite program state $s \circ s' = (\theta \circ \theta', \varphi \wedge \theta(\varphi'), \Xi \circ (\theta \circ \Xi'), l')$, where $\theta(\varphi')$ is a symbolic expression constructed from φ' such that all symbols α_i in φ' are simultaneously substituted by symbolic expressions $\theta(\Theta^{-1}(\alpha_i))$, $\theta \circ \theta'$ is a symbolic memory such that for each variable \mathbf{a} we have $(\theta \circ \theta')(\mathbf{a}) = \theta(\theta'(\mathbf{a}))$, $\theta \circ \Xi' = [\bar{r}'_0, \dots, \bar{r}'_n]$, where each \bar{r}'_i is equal to r_i except the first component being $\bar{r}'_i.\sigma = \theta \circ r'_i.\sigma$, and $\Xi \circ (\theta \circ \Xi') = [r_0, \dots, r_m, \bar{r}'_0, \dots, \bar{r}'_n]$.

Definition 3 (Equivalence) Let φ, φ' be symbolic expressions, θ, θ' be symbolic memories, $\Xi = [r_0, \dots, r_m]$, $\Xi' = [r'_0, \dots, r'_n]$ be call stacks and s, s' be program states. Then $\varphi \equiv \varphi'$, if φ and φ' are either logically equivalent boolean symbolic expressions or numeric symbolic expressions such that $(\varphi = \varphi') \equiv \text{true}$. $\theta \equiv \theta'$, if for each variable \mathbf{a} we have $\theta(\mathbf{a}) \equiv \theta'(\mathbf{a})$. $\Xi \equiv \Xi'$, if $m = n$ and for each $i \in \{0, \dots, m\}$ we have $r_i.\sigma$ and $r'_i.\sigma$ are defined for the same variables with equivalent values and $r_i.l = r'_i.l$. And $s \equiv s'$, if both s and s' have equal or equivalent components.

When returning from a function call, values of local variables are discarded. Therefore, if we have two program states at the same exit location of a function, we may restrict equivalence between symbolic memories of these states only to global variables. Therefore we define also the following equivalence between program states.

Definition 4 (Equivalence on Global Variables) Let s and s' be program states. Then s is equivalent on global variables with s' , written by $s \stackrel{g}{\equiv} s'$, if they have equal or equivalent components except one with symbolic memories, where for each global variable \mathbf{a} we require $s.\theta(\mathbf{a}) \equiv s'.\theta(\mathbf{a})$.

We summarise basic equivalences between composed program states in the following lemma. We do not provide proof since the equivalences is mostly obvious or easy to check.

Lemma 1 (Equivalent Compositions) Let s, s' and s'' be program states, ν and ν' be valuations of all parameters in s and s' respectively such that $\nu \cup \nu'$ is also a valuation, θ, θ' and θ'' be symbolic memories and φ and $\psi \wedge \psi'$ be symbolic

expressions. Then $s \circ (s' \circ s'') \equiv (s \circ s') \circ s''$, $s[\nu] \circ s'[\nu'] \equiv (s \circ s')[\nu \cup \nu']$, $\theta \circ (\theta' \circ \theta'') \equiv (\theta \circ \theta') \circ \theta''$, $(\theta \circ \theta')\langle\varphi\rangle \equiv \theta\langle\theta'\langle\varphi\rangle\rangle$ and $\theta\langle\psi\rangle \wedge \theta\langle\psi'\rangle \equiv \theta\langle\psi \wedge \psi'\rangle$.

Before we formulate a definition of templates with one parameter we give its intuition. Let us consider a part of the program P with an entry location e and n distinct exit locations x_1, \dots, x_n . We saw in Section 2, that key properties for building a template of the part are program states $s_1[\kappa], \dots, s_n[\kappa]$ at exit locations x_1, \dots, x_n . We need to ensure that states $s_i[\kappa]$ correctly represent behaviour of the analysed part. King proved [8] that path conditions at leaf vertices of symbolic execution tree T of P are satisfiable. Therefore, if $s_i.\varphi$ is not satisfiable, then there cannot be a path in T traversing the part from e to x_i . The exit x_i is thus useless for the construction of the template and we omit it. King further showed [8] that for two different leaf vertices u and v of T we have $u.\varphi \wedge v.\varphi \equiv \text{false}$. This statement is also valid for program parts. So, we require $(s_i.\varphi \wedge s_j.\varphi) \equiv \text{false}$ for all different i and j . We summarise these requirements in the following definition.

Definition 5 (Templates with one parameter) Let T be symbolic execution tree of P computed by $*$ -version of Algorithm 1, $n > 0$ be an integer, l, l', l_1, \dots, l_n be locations in P , κ be a parameter, $\theta[\kappa], \theta_1[\kappa], \dots, \theta_n[\kappa]$ be symbolic memories $\varphi_1[\kappa], \dots, \varphi_n[\kappa]$ be satisfiable boolean symbolic expressions such that for each $i, j \in \{1, \dots, n\}, i \neq j$ we have $(\varphi_i \wedge \varphi_j) \equiv \text{false}$ and let $\Xi_1[\kappa], \dots, \Xi_n[\kappa]$ be call stacks.

A tuple $t = (l, n, \{(\theta_1, \varphi_1, \Xi_1, l_1), \dots, (\theta_n, \varphi_n, \Xi_n, l_n)\})$ is a template with one parameter κ in P , if

- (L1) All the locations l, l_1, \dots, l_n in t are neither entry nor exit ones.
- (L2) For each path $\pi = u\omega$ in T from any vertex u satisfying $u.l = t.l$ to a leaf, there is a vertex $w \in \omega$, an index $i \in \{1, \dots, n\}$ and an integer $\nu \geq 0$, such that $w.s \equiv u.s \circ (t.\theta_i, t.\varphi_i, t.\Xi_i, t.l_i)[\nu]$.
- (L3) For each vertex u of T , an index $i \in \{1, \dots, n\}$ and non-negative integer ν such that $u.l = t.l$ and $(u.\varphi \wedge u.\theta\langle t.\varphi_i[\nu]\rangle)$ is satisfiable, there is a successor w of u in T such that $w.s \equiv u.s \circ (t.\theta_i, t.\varphi_i, t.\Xi_i, t.l_i)[\nu]$.

A tuple $t = (l, n, \{(\theta_1, \varphi_1, \Xi_1, l_1), \dots, (\theta_n, \varphi_n, \Xi_n, l_n)\}, \theta, l')$ is a recursion template with one parameter κ in P , if

- (R1) $t.l$ and $t.l'$ are entry and exit locations of the same function respectively and $t.l'$ is the target vertex of an edge with a call action of that function. All the locations l_1, \dots, l_n in t are neither entry nor exit ones.
- (R2) For each path $\pi = u\omega$ in T from any vertex u satisfying $u.l = t.l$ to a leaf, there is a non-leaf vertex $w \in \omega$, an index $i \in \{1, \dots, n\}$ and an integer $\nu \geq 0$, such that $w.s \equiv u.s \circ (t.\theta_i, t.\varphi_i, [(t, \kappa)] \circ t.\Xi_i, t.l_i)[\nu]$.
Further, if there is the first successor \bar{w} of w in π such that $\bar{w}.l = t.l'$ and $\bar{w}.\Xi = w.\Xi$, then there is a non-leaf vertex \bar{u} in a suffix of π starting with \bar{w} such that $\bar{u}.s \stackrel{g}{=} (\bar{w}.\theta \circ t.\theta[\nu], \bar{w}.\varphi, u.\Xi, t.l')$.

(R3) For each vertex u of T , an index $i \in \{1, \dots, n\}$ and non-negative integer ν such that $u.l = t.l$ and $(u.\varphi \wedge u.\theta(t.\varphi_i \llbracket \nu \rrbracket))$ is satisfiable, there is a successor w of u in T such that $w.s \equiv u.s \circ (t.\theta_i, t.\varphi_i, [(t, \kappa)] \circ t.\Xi_i, t.l_i) \llbracket \nu \rrbracket$.

Note that requirements (L2) and (R2) guarantees that no path in T with vertices u and v such that $u.l = l$ and $v.l = l_i$ is suppressed by the state $(t.\theta_i, t.\varphi_i, t.\Xi_i, t.l_i) \llbracket \kappa \rrbracket$. And requirement (L3) and (R3) guarantees that program state $(t.\theta_i, t.\varphi_i, t.\Xi_i, t.l_i) \llbracket \kappa \rrbracket$ does not produce unreal paths. Also note that in requirement (R2) there we use restriction of equivalence to global variables for the phase of returning from recursive calls. Since values of local variables are not important when returning from a function call, the restriction may help to simplify detection of a recursion template.

We are ready to describe \square -version at Algorithm 1. At line 1 there we detect templates with one parameter in the passed program P . That is a task for so called *template detectors*. We discuss a possible construction of such a detector in Section 5. The only purpose of lines 11–31 is to compute successor states of a currently processed program state s . Let us first assume the test at line 11 is *false*. So, we get to line 18. There we call a system function `getTemplatesAt`, which selects those templates, whose entry locations matches the actual program location $s.l$. If the selection is not empty we may instantiate one of the selected templates. A system function `chooseTemplate` is supposed to choose exactly one template t to be instantiated. We may for example choose randomly. We do not put any constraints to the selection strategy. To prevent parameter collisions we first get a fresh one at line 21 and then we replace the parameter used in t by default by the fresh one. Now we have two possibilities. Either t is a recursion template or not. In the first case we get to a loop at line 24. There we create $t.n$ successors of the program state s (see line 25). Note that call stack of i -th successor state is of the form $s.\Xi \circ [(t, \kappa)] \circ t.\Xi_i$. It means that the special record is at the position in the stack, when we entered the recursive function. The only special record (t, κ) in the call stack represents *any* possible number of subsequent recursive calls in classic symbolic execution. The record also saves reference to the template t and the parameter κ for the later phase of returning from the recursive calls. If t is not a recursion template, then it must be our general purpose template with one parameter (since we do not consider any other kinds of templates in this paper). So we get to line 28 in the algorithm. There we also create successors of the program state s (see line 29). It remains to discuss the computation of successors, when the condition at line 11 is *true*. The condition says that the location $s.l$ references exit location of a function and that there is the special record (t, κ) at the top of the call stack $s.\Xi$. In other words, we reached the moment, when we have to return from recursive calls. We first retrieve the recursive template and the parameter used in the instantiation of t (see lines 12 and 13). After substitution of the default parameter by the retrieved one, we finish the instantiation of t by computing the only successor of the actual state. The successor state represents the effect of all the returns from recursive calls done previously. This is ensured by using of the same parameter form both phases of the instantiation of the template t .

A number of recursive calls therefore matches the number of returns from them. Also note that call stack of the successor does not contain the special record. We finish the description of the algorithm by the following observation. The expressions computing successor states at lines 15, 25 and 29 precisely match corresponding expressions in Definition 5. Note that at line 15 there the call stack $\text{pop}(s.\Xi)$ must be equal to one of a program state, for which we previously get to line 25. And this program state had to be related to the entry location of a function causing the recursive calls.

4 Soundness and Completeness

In this section we formulate and prove soundness and completeness theorems for compact symbolic execution using recursive and general templates with one parameter. The theorems say that both classic and compact symbolic execution explore the same set of real paths of P . To avoid repetitions we assume for the remainder of this section that P is a program, and T and T' are symbolic execution trees of the program P computed by $*$ - and \square , $*$ -versions of Algorithm 1 respectively.

Lemma 2 *Call stack records pushed at line 25 of Algorithm 1 cannot be adjacent in call stacks of vertices of T' .*

Proof. Follows immediately from requirement for locations of templates in Definition 5 and from the fact, that reaching line 25 requires a processed state must reference a function entry location.

Lemma 3 *Let $u \in T$, $u' \in T'$, $u'.\Xi \neq []$, $\text{top}(u'.\Xi) = (t, \kappa)$, $u'.l$ is an exit location and $u.s \stackrel{g}{\equiv} u'.s[\nu]$ for some valuation ν . Then there are the only direct successors $w \in T$ and $w' \in T'$ of u and u' respectively and they satisfy $w.s \equiv w'.s[\nu]$.*

Proof. Follows directly from Lemma 2 and from the fact that successors of u' are computed at line 32 of Algorithm 1.

Theorem 1 (Soundness) *For each leaf vertex $e \in T$ there is a leaf vertex $e' \in T'$ and a valuation ν of all parameters in $e'.s$ such that $e.s \equiv e'.s[\nu]$.*

Proof. Let π be the path in T from the root to the leaf vertex e . We prove the theorem by the following induction:

Basic case: The root vertices r and r' of T and T' respectively are labelled by the same program state s_0 (see lines 2 and 4). So, $r.s \equiv r'.s[\nu]$, for $\nu = \emptyset$.

Inductive step: Let $u \in \pi$, $u \neq e$, u' be a vertex of T' and ν be a valuation such that $u.s \equiv u'.s[\nu]$. We show, there is a successor w of u in π , a successor vertex w' of u' in T' and a valuation ν' such that $w.s \equiv w'.s[\nu']$. And we further show there is no vertex v' in the path between u' and w' in T' such that successors of $v'.s$ are computed at line 25. There are four possible cases in Algorithm 1 for $u'.s$:

(1) We reach line 28: According to Definition 5 (L2), there is a successor vertex w of u in π , an index i and a non-negative integer ν for κ such that

$$\begin{aligned} w.s &\equiv u.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i) \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\equiv u'.s \llbracket \nu \rrbracket \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i) \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\equiv (u'.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i)) \llbracket \nu \cup \{(\kappa, \nu)\} \rrbracket \\ &\equiv s' \llbracket \nu' \rrbracket, \end{aligned}$$

where s' is the i -th direct successor of $u'.s$ computed at line 29. And since $w \in T$, we have $w'.\varphi$ is satisfiable. Therefore, there is be a direct successor w' of u' in T' with $w.s = s'$.

(2) We reach line 24: According to Definition 5 (R2), there is a successor vertex w of u in π , an index i and a non-negative integer ν for κ such that

$$\begin{aligned} w.s &\equiv u.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i) \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\equiv u'.s \llbracket \nu \rrbracket \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i) \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\equiv (u'.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i)) \llbracket \nu \cup \{(\kappa, \nu)\} \rrbracket, \\ &\equiv s' \llbracket \nu' \rrbracket, \end{aligned}$$

where s' is the i -th direct successor of $u'.s$ computed at line 25. And since $w \in T$, we have $w'.\varphi$ is satisfiable. Therefore, there is a direct successor w' of u' in T' with $w.s = s'$.

(3) We reach line 12: Let π' be a path in T' from the root to the vertex u' . According to connections between vertices u' constructed for vertices u along π , there is a predecessor x' of u' in π' , which pushed (at line 25) the record being at the top of $u'.\Xi$. Obviously, successors of $x'.s$ are computed at line 25. Therefore, there is $x \in \pi$ such that $x.s \equiv x'.s \llbracket \nu \rrbracket$. According to case (2) there is a successor y of x in π and a direct successor y' of x' in π' such that $y.s \equiv y'.s \llbracket \nu \rrbracket$. Note that $y'.s$ uses the parameter κ retrieved from stack $u'.\Xi$ at line 13. Therefore, valuation ν defines an integer $\nu = \nu(\kappa)$. Also note that u is the first successor of y in π with $u.l$ being an exit location and $u.\Xi = y.\Xi$. Otherwise we would apply this case (3) for some other vertex lying between y' and u' in π' . Therefore, from Definition 5 (R2) there is a non-leaf vertex v in a suffix of π starting with u such that

$$\begin{aligned} v.s &\stackrel{g}{\equiv} (u.\theta \circ t.\theta[\kappa], u.\varphi, x.\Xi, t.l') \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\stackrel{g}{\equiv} (u'.\theta \llbracket \nu \rrbracket \circ t.\theta[\kappa], u'.\varphi \llbracket \nu \rrbracket, \text{pop}(u'.\Xi) \llbracket \nu \rrbracket, t.l') \llbracket \{(\kappa, \nu)\} \rrbracket \\ &\stackrel{g}{\equiv} (u'.\theta \circ t.\theta[\kappa], u'.\varphi, \text{pop}(u'.\Xi), t.l') \llbracket \nu \rrbracket \\ &\stackrel{g}{\equiv} s' \llbracket \nu \rrbracket, \end{aligned}$$

where s' is the only successor state of $u'.s$ computed at line 15. Since $v \in T$, then $s'.\varphi$ is satisfiable and there is a direct successor v' of u' in T' with $v'.s = s'$. And finally Lemma 3 ensures there are the only direct successors w and w' of v and v' respectively, such that $w.s \equiv w'.s \llbracket \nu \rrbracket$.

(4) Otherwise, we reach line 32: Since $u.s \equiv u'.s[\nu]$ and we apply classic symbolic execution step for $u'.s$, there must be a direct successor w of u and a direct successor w' of u' such that $w.s \equiv w'.s[\nu]$.

Theorem 2 (Completeness) *For each leaf vertex $e' \in T'$ there is a leaf vertex $e \in T$ and a valuation ν of all parameters in $e'.s$ such that $e.s = e'.s[\nu]$.*

Proof. Let π' be the path in T' from the root to the leaf vertex e' . We prove the theorem by the following induction:

Basic case: The root vertices r and r' of T and T' respectively are labelled by the same program state s_0 (see lines 2 and 4). Let us construct a non-empty set U of vertices of T such that for each valuation ν of all parameters in $r'.s$ such that $r'.\varphi[\nu]$ is satisfiable, there is $u \in U$ such that $u.s \equiv r'.s[\nu]$. Obviously $U = \{r\}$, because $r'.\varphi$ contains no parameter (so $r.s \equiv r'.s[\nu]$, for each ν).

Inductive step: Let $u' \in \pi'$, $u' \neq e'$ and U be a non-empty set of vertices of T such that for each valuation ν of all parameters in $u'.s$ such that $u'.\varphi[\nu]$ is satisfiable, there is $u \in U$ such that $u.s \equiv u'.s[\nu]$. We show, there is a successor w' of u' in π' and a non-empty set W of vertices of T such that for each valuation ν' of all parameters in $w'.s$ such that $w'.\varphi[\nu']$ is satisfiable, there is $w \in W$ such that $w.s \equiv w'.s[\nu']$. And we further show that each $w \in W$ is a successor of some $u \in U$ and there is no vertex v' between u' and w' in π' such that successors of $v'.s$ are computed at line 25. There are four possible cases in Algorithm 1 for $u'.s$:

(1) We reach line 28: Let w' be a direct successor of u' in π' . Obviously, $w'.s$ is one of the states s' computed at line 29. Let i be the index, for which $w'.s = s'$. The formula $w'.\varphi$ is satisfiable, since w' is in T' (see condition at line 34). Let ν be a valuation for which $w'.\varphi$ is satisfiable. And let $\nu' = \nu \setminus \{(\kappa, \nu)\}$, where ν is an integer assigned in ν to the fresh parameter κ introduced at line 21. From line 29 we see that $u'.\varphi[\nu']$ is satisfiable. Therefore, there is a vertex $u \in U$ such that $u.s \equiv u'.s[\nu']$. According to Definition 5 (L3) there is a successor w of u in T such that

$$\begin{aligned} w.s &\equiv u.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i)[\{(\kappa, \nu)\}] \\ &\equiv u'.s[\nu'] \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i)[\{(\kappa, \nu)\}] \\ &\equiv (u'.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], t.\Xi_i[\kappa], t.l_i))[\nu] \\ &\equiv w'.s[\nu]. \end{aligned}$$

Therefore, $w \in W$.

(2) We reach line 24: Let w' be a direct successor of u' in π' . Obviously, $w'.s$ is one of the states s' computed at line 25. Let i be the index, for which $w'.s = s'$. The formula $w'.\varphi$ is satisfiable, since w' is in T' (see condition at line 34). Let ν be a valuation for which $w'.\varphi$ is satisfiable. And let $\nu' = \nu \setminus \{(\kappa, \nu)\}$, where ν is an integer assigned in ν to the fresh parameter κ introduced at line 21. From line 25 we see that $u'.\varphi[\nu']$ is satisfiable. Therefore, there is a vertex $u \in U$ such that $u.s \equiv u'.s[\nu']$. According to Definition 5 (R3) there is a successor w of u

in T such that

$$\begin{aligned}
w.s &\equiv u.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i)[\{(\kappa, \nu)\}] \\
&\equiv u'.s[\nu'] \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i)[\{(\kappa, \nu)\}] \\
&\equiv (u'.s \circ (t.\theta_i[\kappa], t.\varphi_i[\kappa], [(t, \kappa)] \circ t.\Xi_i[\kappa], t.l_i))[\nu] \\
&\equiv w'.s[\nu].
\end{aligned}$$

Therefore, $w \in W$.

(3) We reach line 12: Let x' be a predecessor of u' in π' , which pushed (at line 25) the record being at the top of $u'.\Xi$. Obviously, successors of $x'.s$ are computed at line 25. Further, let y' and v' be direct successors of x' and u' in π' respectively. The formula $v'.\varphi$ is satisfiable, since v' is in T' (see condition at line 34). Note that v' is the only successor of u' in T' . Let ν be a valuation for which $v'.\varphi$ is satisfiable. Note that ν defines an integer $\nu = \nu(\kappa)$ for the parameter κ retrieved from stack $u'.\Xi$ at line 13, since $y'.s$ must have already used it. From line 15 we see that $u'.\varphi[\nu]$ is satisfiable. Therefore, there is a vertex $u \in U$ such that $u.s \equiv u'.s[\nu]$. Let π be a path in T from the root to a leaf vertex and going through u . According to connections between vertices of sets U constructed for vertices u' along π' , there is a predecessor x of u in π , such that $x.s \equiv x'.s[\nu]$. Since y' is the direct successor of x in π (i.e. there was computed a set W for y'), there must also exist a vertex $y \in \pi$ lying between x and u and $y.s \equiv y'.s[\nu]$. Note that u is the first successor of y in π with $u.l$ being an exit location and $u.\Xi = y.\Xi$. Otherwise we would apply this case (3) for some other vertex lying between y' and u' in π' . Therefore, from Definition 5 (R2) there is a non-leaf vertex v in a suffix of π starting with u such that

$$\begin{aligned}
v.s &\stackrel{g}{\equiv} (u.\theta \circ t.\theta[\kappa], u.\varphi, x.\Xi, t.l')[\{(\kappa, \nu)\}] \\
&\stackrel{g}{\equiv} (u'.\theta[\nu] \circ t.\theta[\kappa], u'.\varphi[\nu], \text{pop}(u'.\Xi)[\nu], t.l')[\{(\kappa, \nu)\}] \\
&\stackrel{g}{\equiv} (u'.\theta \circ t.\theta[\kappa], u'.\varphi, \text{pop}(u'.\Xi), t.l')[\nu] \\
&\stackrel{g}{\equiv} v'.s[\nu].
\end{aligned}$$

And finally Lemma 3 ensures there are the only direct successors w and w' of v and v' respectively, such that $w.s \equiv w'.s[\nu]$. Therefore, $w \in W$.

(4) Otherwise, we reach line 32: Let u be any vertex in U . Since $u.s \equiv u'.s[\nu]$ for some valuation ν for which $u'.\varphi[\nu]$ is satisfiable and since all direct successors of both u and u' are computed by classic symbolic execution step, there must be a direct successor w of u in T and a direct successor w' of u' in T' such that $w.s \equiv w'.s[\nu]$. Note that both $u'.s$ and $w'.s$ have exactly the same parameters. Therefore, $w \in W$.

5 Computation of Templates

In this section we show one possible approach to computation of templates with one parameter. We provide detailed description of an algorithm computing a

template for a program part with specified cyclic path, entry location, and several exit ones. Then we extend concept of the algorithm to computation of recursion templates for program parts.

5.1 Template for Program Part with Cyclic Path

Let P be a program and let us suppose we have a program part of P with a cyclic path, an entry location e and some exit location x (but there can be other exits from the part). We show how to compute a symbolic memory $\theta_x[\kappa]$, a path condition $\varphi_x[\kappa]$ and a call stack $\Xi_x[\kappa]$ at the exit location x . The computation of remaining parts of resulting template are then straightforward.

The algorithm proceeds in two steps. First, we compute a program state $(\theta, \varphi, [], e)$ resulting from classic symbolic execution of the cyclic path of the part exactly once, and a program state $(\hat{\theta}, \hat{\varphi}, \hat{\Xi}, x)$ resulting from classic symbolic execution of a path from e to x . The second step is to express $\theta_x[\kappa]$, $\varphi_x[\kappa]$ and $\Xi_x[\kappa]$ in terms of the program states computed in the first step.

The computation of program states $(\theta, \varphi, [], e)$ and $(\hat{\theta}, \hat{\varphi}, \hat{\Xi}, x)$ requires to run classic symbolic execution on the analysed program part. But Algorithm 1 can only execute programs satisfying Definition 1. Therefore, we create a new program, say P' , representing the analysed part.

We start with a program P' consisting of all variables of P and of all those functions of P having at least one location of the cycle. Note that the cyclic path of the part may traverse several functions through call sites. We now remove all the locations and edges in P' , which do not belong to the cycle nor to the path from e to x . We assume that x does not belong to the cyclic path, since otherwise we can always create its copy outside the cycle. Next we mark the function in P' containing the entry location e as the starting function of P' and we set e to be the entry location of the function. Then we create a new location e' representing the exit location from the starting function. Now we break the cyclic path in the entry e such that we redirect the only in-edge of e (belonging to the cycle) to e' . And finally we transform x to error location by adding loop edge with **skip** action.

P' is now a program according to Definition 1. So, we can run unmarked version of Algorithm 1. Note that the algorithm must always terminate for P' . Let E be a set of resulting program states. Then $|E| \leq 2$. If there is no $s \in E$ such that $s.l = e$, then we do not create the template for the part, since there is no real path around the cycle. If there is no state $s \in E$ such that $s.l = x$, then we discard the exit x from the consideration for the template, since it is impossible to leave the loop through x . Otherwise, E contains exactly two program states, which are the states we are looking for.

Now we show how to express $\theta_x[\kappa]$, $\varphi_x[\kappa]$ and $\Xi_x[\kappa]$ in terms of the program states computed above. Let T be a symbolic execution tree of P , computed by *-version of Algorithm 1. Further, let u be a vertex of T such that $u.l = e$ and $\pi = u \dots u_1 \dots u_2 \dots u_\nu \dots w$ be a path in T starting at u , iterating the cycle of the part exactly $\nu \geq 0$ times, i.e. all the vertices u_i have $u_i.l = e$, and then π

leaves the cycle into the vertex w with $w.l = x$. We use memory composition to express memories of vertices along π as follows.

$$\begin{aligned} u_1.\theta &= u.\theta \circ \theta \\ u_2.\theta &= u_1.\theta \circ \theta = u.\theta \circ (\theta \circ \theta) \\ &\dots \\ u_\nu.\theta &= u_{\nu-1}.\theta \circ \theta = u.\theta \circ \underbrace{(\theta \circ \dots \circ \theta)}_\nu. \end{aligned}$$

If we denote the composition of i symbolic memories θ by θ^i , where $\theta^0 = \Theta$ and $\theta^1 = \theta$, then we have $u_i.\theta = u.\theta \circ \theta^i$ and we get

$$w.\theta = u.\theta \circ (\theta^\nu \circ \hat{\theta}).$$

We proceed similarly to express path conditions of vertices along π .

$$\begin{aligned} u_1.\varphi &\equiv u.\varphi \wedge u.\theta\langle\varphi\rangle \equiv u.\varphi \wedge (u.\theta \circ \theta^0)\langle\varphi\rangle \equiv u.\varphi \wedge u.\theta\langle\theta^0\langle\varphi\rangle\rangle \\ u_2.\varphi &\equiv u_1.\varphi \wedge u_1.\theta\langle\varphi\rangle \equiv u.\varphi \wedge u.\theta\langle\theta^0\langle\varphi\rangle\rangle \wedge (u.\theta \circ \theta^1)\langle\varphi\rangle \equiv u.\varphi \wedge u.\theta\langle\theta^0\langle\varphi\rangle \wedge \theta^1\langle\varphi\rangle\rangle \\ &\dots \\ u_\nu.\varphi &\equiv u_{\nu-1}.\varphi \wedge u_{\nu-1}.\theta\langle\varphi\rangle \equiv u.\varphi \wedge u.\theta\langle\underbrace{\theta^0\langle\varphi\rangle \wedge \dots \wedge \theta^{\nu-1}\langle\varphi\rangle}_\nu\rangle \end{aligned}$$

Using the following equivalence

$$\theta^0\langle\varphi\rangle \wedge \dots \wedge \theta^{\nu-1}\langle\varphi\rangle \equiv 0 \leq \nu \wedge \forall \tau \ (0 \leq \tau < \nu \rightarrow \theta^\tau\langle\varphi\rangle),$$

we can write

$$\begin{aligned} w.\varphi &\equiv u_\nu.\varphi \wedge u_\nu.\theta\langle\hat{\varphi}\rangle \equiv u.\varphi \wedge u.\theta\langle\theta^0\langle\varphi\rangle \wedge \dots \wedge \theta^{\nu-1}\langle\varphi\rangle \wedge \theta^\nu\langle\hat{\varphi}\rangle\rangle \\ &\equiv u.\varphi \wedge u.\theta\langle 0 \leq \nu \wedge \forall \tau \ (0 \leq \tau < \nu \rightarrow \theta^\tau\langle\varphi\rangle) \wedge \theta^\nu\langle\hat{\varphi}\rangle \rangle. \end{aligned}$$

SMT solvers do not support memory composition operation appearing in the formula $w.\varphi$. Therefore, we need an equivalent *declarative* description of the operation. Such a description is a parametrised symbolic memory $\theta\llbracket\kappa\rrbracket$, where we require $\theta\llbracket\kappa\rrbracket \equiv \theta^\kappa$, for any $\kappa \geq 0$. For a given symbolic memory θ we compute content of $\theta\llbracket\kappa\rrbracket$ per variable by applying the following two rules

$$\frac{\theta(\mathbf{a}) = \Theta(\mathbf{a}) + c, \quad \mathbf{a} \text{ is of a numeric type, } c \text{ is a numeric constant of } \mathbf{a}'\text{'s type}}{\theta\llbracket\kappa\rrbracket(\mathbf{a}) = \Theta(\mathbf{a}) + c \cdot \text{typeOf}\langle\mathbf{a}\rangle(\kappa)},$$

$$\frac{\theta(\mathbf{A}) = \Theta(\mathbf{A}), \quad \mathbf{A} \text{ is of a an array type}}{\theta\llbracket\kappa\rrbracket(\mathbf{A}) = \Theta(\mathbf{A})},$$

where expression $\text{typeOf}\langle\mathbf{a}\rangle(\kappa)$ represent casting operation of κ to a type of variable \mathbf{a} . If there is a variable, which does not match any of the rules, then we fail to compute $\theta\llbracket\kappa\rrbracket$. And we thus fail to compute the template. Obviously,

one can provide more rules for more complex symbolic memories. The presented rules are only supposed to illustrate the process.

Having $\theta[\kappa]$ we define

$$\begin{aligned}\theta_x[\kappa] &= \theta[\kappa] \circ \hat{\theta} \\ \varphi_x[\kappa] &= 0 \leq \kappa \wedge \forall \tau \ (0 \leq \tau < \kappa \rightarrow \theta[\tau] \langle \varphi \rangle) \wedge \theta[\kappa] \langle \hat{\varphi} \rangle \\ \Xi_x[\kappa] &= \theta[\kappa] \circ \hat{\Xi},\end{aligned}$$

and we get $w.\theta \equiv u.\theta \circ \theta_x[\nu]$, $w.\varphi \equiv u.\varphi \wedge u.\theta \langle \varphi_x[\nu] \rangle$ and $w.\Xi \equiv u.\Xi \circ (u.\theta \circ \Xi_x[\nu])$. Using these equivalences we write $w.s \equiv u.s \circ (\theta_x, \varphi_x, \Xi_x, x)[\nu]$, which is exactly the equivalence used in Definition 5 (L2) and (L3).

5.2 Template for Program Parts Representing Recursion

Let P be a program, f be a recursive function of P , e and x be entry and exit locations of f respectively and let $h = (u, v)$ be an edge of f with an action representing recursive call of f . We transform computation of recursion template for recursive calling of f into analysis of two program parts P_1 and P_2 with cyclic paths. The cycle of P_1 starts at location e and leads to u . We then enclose the cycle by an artificial edge whose action simulate an effect of any call of f . Let e be entry location of P_1 and let x_1, \dots, x_n be its exit locations. We compute a template $t_1 = (e, n, \{(\theta_1, \varphi_1, \Xi_1, x_1)[\kappa], \dots, (\theta_n, \varphi_n, \Xi_n, x_n)[\kappa]\})$ for P_1 according to algorithm from Section 5.1. Having t_1 we can express the resulting recursive template t as follows.

$$t = (e, n, \{(\theta_1, \varphi_1, \Xi_1, x_1)[\kappa], \dots, (\theta_n, \varphi_n, \Xi_n, x_n)[\kappa]\}, \theta[\kappa], x),$$

where $\theta[\kappa]$ is the only unknown component in t . We compute the symbolic memory θ from analysis of the second program part P_2 . The cycle of P_2 starts at x . There we add an artificial edge, whose action simulate an effect of return from any call of f . The artificial edge gets us to location v . Then we enclose the cycle by following a path from v to x . We set x to be the entry location of P_2 and we further set x to also be the only exit location from P_2 . As you can see, here we have introduced an assumption that there is no branching along the path from v to x , i.e. we cannot escape from the path. We discuss the case, when there is some branching (escape edges) along the path later. Since we have defined the program part P_2 , we compute its template $t_2 = (x, 1, \{(\theta[\kappa], \text{true}, [], x)\})$ according to algorithm from Section 5.1. Then we take the symbolic memory $\theta[\kappa]$ and we complete the recursion template t .

Note that we can simplify computation of $\theta[\kappa]$ of the template t_2 such that we only express a return value of f . We do not need to express local variables of f , since requirement (R2) of Definition 5 uses the equivalence $\stackrel{g}{\equiv}$. We further note, that the algorithm above also works for indirect recursion. It immediately follows from the algorithm in Section 5.1, where cyclic path of an analysed program part may traverse several functions.

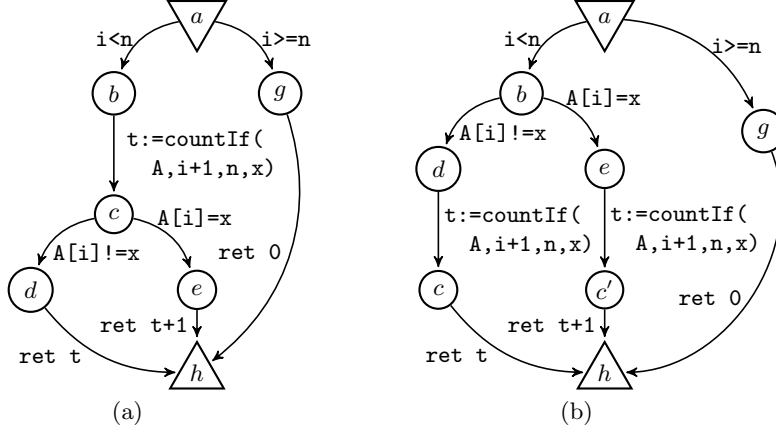


Fig. 4. Two equivalent recursive implementations of the function `countIf(A, i, n, x)`.

We finish the section by discussion of the assumption we gave to the cyclic path of P_2 . We assumed there is no branching along the path from v to x . The algorithm presented above can compute templates for tail recursions and for many non-tail ones, while keeping the computation simple (we only need $\theta[\kappa]$ expressed just for return value). Therefore, we believe the assumption has only small impact to applicability of the algorithm. Besides, it is always possible to move edges with recursive calls below branchings not depending on return values from the calls. We demonstrate this process at Figures 4 (a) and (b), where we depict two equivalent recursive implementations of the function `countIf`. We can easily check that in program at Figures 4 (b) there are two program parts (one per recursive call), for which we can compute templates according to the algorithm described above.

6 Discussion

We presented compact symbolic execution using only templates with a single parameter. We further restrict ourselves to computation of templates only for program parts consisting of cyclic paths of representing recursion. We can get even better reduction of size of symbolic execution tree, if we create templates for more complex program parts, and when we use more parameters. Let us consider the function `countIf` at Figure 2. The program loop in the function consists of two cyclic paths around it. We have already discussed templates for both cycles in Section 2. But if we built a single template using two parameters (one parameter per cyclic path), then resulting compact symbolic execution tree would be finite. We see, there is a space for extensions of the basic concepts we presented here.

Let us consider well known algorithm `binarySearch`. Template detection for this program (even with a single parameter) may infer geometric progressions

as values of some variables. They may later cause serious performance issues for SMT solver, when they get into a path condition.

Compact symbolic execution commonly has higher performance requirements to SMT solvers than classic one. Path conditions may contain template parameters besides symbols. And parameters are quantified. This is the price of the ability to reason about multiple program paths at once.

King showed effectiveness of symbolic execution for automated testing generation [8]. Producing a good test typically means to reach some interesting (e.g. bug suspicious) program location. Compact symbolic execution can be very helpful in this task. Let us consider a situation, when reachability of such a target location is dependant on an exact number of iterations of a particular cycle. Providing a template for a program part with the cycle, we can simultaneously reason about all the paths exiting from the cycle. Therefore, instead of exploration of paths space by classic symbolic execution, we can just send a query to SMT solver to check satisfiability of parametrised path condition.

King also showed in his paper [8], how symbolic execution can be used in proving program correctness according to Floyd's method [3]. Using templates we can decrease or in some cases even eliminate the need of loop invariants. For programs, where compact symbolic execution is finite in contrast to classic one, there we do not need loop invariants at all. And for other programs, loop templates describe behaviour of some paths through loop, and we may therefore provide simpler invariants for the remaining behaviour of the loop.

7 Related Work

Compact symbolic execution is tightly related to the work of King in 1976 [8], where the author introduced the general concept of classic symbolic execution. Besides the description of symbolic execution King discussed its applicability to program testing and formal proving of correctness according to Floyd's method [3]. Nevertheless, issues like the path explosion problem were not tackled.

In [6] authors propose a program instrumentation by a code providing lazy initialisation of dynamically allocated data structures like lists or trees and they enable symbolic execution of the instrumented program by a standard model checker without building a dedicated tool. The lazy initialisation algorithm is further improved and formally defined as an operational semantics of a core subset of the Java Virtual Machine in [2].

A scalability of symbolic execution to real world programs can be improved by exploring only client's code [7]. A library code (like string manipulation, standard containers like sets or maps) can be assumed as well defined and properly tested.

There are several symbolic execution based techniques constructing loop summaries or simply counting loop iterations [5,11,12]. The introduction of counters usually provides a possibility to speak about multiple paths through loop at once. A technique presented in [5] 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 and postconditions. The LESE technique presented in [11] introduces symbolic variables for the number of times each loop was executed. 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 symbolic-execution-based algorithm in [12] produces a nontrivial necessary condition on input values to drive the program execution to the given location. The key part of the technique is computation of loop summaries in form of symbolic program states and path conditions both parametrised by so called path counters. Each path counter is assigned to individual path through the analysed loop.

There are also approaches computing function summaries [4,1]. Reusing summaries at call sites typically leads to an interesting performance improvement. Moreover, summaries may insert additional symbolic values into a path condition which often leads to another performance improvement.

Finally, there are also techniques partitioning program paths into separate classes according to impact of the paths to a given set of program variables [9,10]. Values of output variables are typically considered as a partitioning criteria.

8 Conclusion

We introduced a generalisation of classic symbolic execution called compact symbolic execution. We generalised notion of symbols of classic symbolic execution such that symbols can be related to different program locations now. This allows us to analyse individual parts of a given program separately from the rest of the program. We further introduced concept of templates representing declarative parametric descriptions of behaviour of separately analysed program parts. We gave precise definition of templates with one parameter and we provided algorithm of compact symbolic execution using these templates.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08*, pages 367–381. Springer, 2008.
2. X. Deng, J. Lee, and Robby. Efficient and formal generalized symbolic execution. *Automated Software Engineering*, pages 1–69, 2011.
3. R. W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, pages 19–31, 1967.
4. P. Godefroid. Compositional dynamic test generation. In *POPL '07*, pages 47–54. ACM, 2007.
5. P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA '11*, pages 23–33. ACM, 2011.
6. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, pages 553–568. Springer-Verlag, 2003.
7. S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In *PASTE '05*, pages 103–110. ACM, 2005.

8. J. C. King. Symbolic execution and program testing. *Commun. ACM*, pages 385–394, 1976.
9. D. Qi, H. D. T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *ESEC/FSE '11*, pages 278–288. ACM, 2011.
10. R. A. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA 2010*, pages 195–206. ACM, 2010.
11. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA '09*, pages 225–236. ACM, 2009.
12. J. Strejček and M. Trtík. Abstracting path conditions. *arXiv.org*, 2011. <http://arxiv.org/abs/1112.5671>.