Improving Scalability of Compositional Symbolic Execution through Parameterized Summary

Tong Xue[1] Shi Ying[1] Xiang-Yang Jia[1*]

[1] State Key Lab of Software Engineering, Wuhan University

Wuhan, China

{xuetong, jxy, yingshi}@whu.edu.cn

[*] Corresponding author

Received 29 July 2015; Revised 1 September 2015; Accepted 1 December 2015

Abstract. Summary-based compositional symbolic execution is able to merge the paths of target method into one logic formula, so that to eliminate path explosion due to inter-procedural paths. However, constraints in such summary-base method are always more complex than classical symbolic execution. When the constraints are complex enough, it will cost a lot of time in constraint solving, so as lead to worse scalability. In order to improve the scalability of compositional symbolic execution, we propose a more flexible summary-based approach named DEMPS, which introduces parameterization mechanism into summary, so that a summary specification is able to be instantiated with different calling context, as well as to be composed with each other in hierarchical way. DEMPS improves the scalability though on-demand path expanding and merging of summary. On-demand expanding selectively expands the paths of summary instances which contribute to the target, and keeps the useless summary instances abstract. On-demand merging is able to selectively merge or expand paths of specific summary instances, so as to support the trade-off between path exploration and constraint solving. We implement DEMPS in Symbolic JPF, and experimental results show that, when we choose a reasonable decision of merging and expanding, DEMPS has better scalability than both classical symbolic execution and classical summary-based approach.

Keywords: symbolic execution, scalability, summary

1 Introduction

Symbolic execution is a well-known program analysis technique, which proved to be practical for automated test case generation and bug finding [1]. Some tools, e.g. CUTE [2], KLEE [3], symbolic JPF [4], have been widely used in industrial software development. Symbolic execution works by exploring as many program paths as possible in a given time budget, creating logical formula encoding the explored paths, using a constraint solver to generate test inputs for feasible execution paths, as well as finding corner-case bugs such as buffer overflows, uncaught exceptions, or even higher-level program assertions [5,6].

However, symbolic execution always suffers from challenge problems of path explosion and complex constraint solving [5,6,7]. A lot of techniques are proposed to cope with the path explosion, such as heuristic search[8], interleaving random and symbolic execution [9], paths pruning [10,11,12], lazy test generation [13], static path merging [14,15,16], incremental/regressive symbolic execution [17,18,28], combining static checking and dynamic symbolic execution [19], etc. Constraint solving is also able to be eased by techniques like irrelevant constraint elimination and incremental solving [20,21].

Compositional symbolic execution is a group of approaches, which cope with the path exploration issues based on summary [20,21,22]. The summary of method is specified as a formula of propositional logic, which merges the pre/post condition of the method paths together with disjunction. By reusing the summary, compositional symbolic execution treats method calling as one state, instead of exploring the method paths. Therefore, it is able to eliminate path explosion due to inter-procedural (inter-block) paths.

However, through the mitigating of path explosion by summary-based approach, it increases the complexity of constraints [6,7,16]. Method summary merges multiple constraints of paths in summarized method into one formula, which will be composed into caller's path conditions, therefore result in longer and more complex path constraints than that of classical symbolic execution. When the constraints are complex enough, it will cost a lot of time in constraint solving, so as lead to worse scalability.

In order to overcome this problem, we propose a more flexible summary-based approach named DEMPS (on-Demand Expanding and Merging of Parameterized Summary), which introduces parameterization mechanism into summary, so that a summary specification is able to be instantiated with different calling context. Just like

the function in programming language, parameterized summaries are able to be composed with each other in hierarchical style.

The parameterized summary improves the scalability though on-demand path expanding and merging. On-demand expanding eases the path exploration, in that it only expands the paths of summary instances which contribute to the target, and keeps the useless summary instances abstract. On-demand merging is helpful for trade-off between path exploration and constraint solving, so that we can get reasonable complexity of constraint at a cost of acceptable path exploration by merging or expanding the paths of specific summary instances.

We have implemented DEMPS in Symbolic JPF, and experimental results show that, when we choose a reasonable decision of merging and expanding, DEMPS has better scalability than both classical symbolic execution and classical summary-based approach.

2 Parameterized Summary

The idea of parameterized summary is similar to the function in programming language. The summary of method or code fragment has a list of input variables and a pre-condition as its parameters, which can be assigned with symbolic values and constraint formula in the calling context.

Just like the function in programming language, the parameterized summaries in program will be composed in hierarchal style. One summary specification can compose other summaries into its path, in the form of abstract summary instance expressions.

2.1 Definition of Parameterized Summary

Definition 1: (Parameterized Summary) *S=(I, O, C, PS), where I is the value list of Input valuables, O is the set of output valuables, C is the pre-condition of the summary, PS is the path set of program. For every p* ϵ *PS, p=(lpc,T), where lpc is the local path condition formulas for this path. T is the trace of this path, including a sequence of path steps.*

Definition 2: (Path Step) *Path step specifies the action in the path. The step set STEP ={s|* $s \in DW \cup EP \cup SI$*}, where* $DW \subset (O \cup Output(SI)) \times E$ *, is the set of data-written steps, output(SI) is a set of output for summary instance set SI, and E is the symbolic expression set; EP is the set of path-set step, which contains a set of embedded sub-paths; SI is the set of summary instance step, which contains a summary instance.*

Definition 3: (Summary Instance) *Summary instance is defined as a triple tuple:* $SI=<S_{si}, I_{si}, C_{si}>$*, where i.e.* S_{si} *is the referred summary of this instance,* I_{si} *is the symbolic values of the input list, and* C_{si} *is a unquantified first logic formula of pre-condition.*

In these definitions, the input I and pre-condition C play the role of parameters, which will be assigned with symbolic values in summary instances.

The output of the summary specifies the summary's effect on its callers. The outputs include the return value, the modified field of this object, the modified field of static class, as well as the modified input objects.

The path set P of the summary is the body of summary. The local path condition lpc of path specifies the path condition without pre-constraint passed on from predecessor states. The path trace specifies the observed behavior of program in specific path, i.e. it only records the actions that may have effects on its caller.

The path set of the summary specifies the main structure of symbolic execution tree of target method or code fragment. The definition of path step shows that the paths are organized in hierarchal style, i.e. the paths in path set might have embedded path sets, as well as the embedded summary instances that can be expanded as path sets.

2.2 Examples

We use an example to show how parameterized summaries are specified. The example Java code and its summary specifications are shown as Fig. 1.

In these two summary specifications, the input variables are parameters of method, output variable is the method return *ret*, the pre-condition are set as true. In the path-set of summary *foo(x,y)*, *bar*(y_0) is a summary instance step, which has a input value y_0, and a pre-condition $x_0>0$ *; ret=bar*(y_0)*.ret* is a data-written step, which assign the *bar(y).ret(*one output of summary instance *bar(y))* to ret *; {*($y_0>0,\ldots$),($y_0<=0,\ldots$)*}* is a path-set step that embedded into the path ($x_0<=0,\ldots$).

The parameterized summary can be applied both method and code fragment. If it is applied to the code fragment, the input will be the values of variables that defined before the fragment, and output will be the variables that used after the fragment.

```
Example Code 1:
int bar(int m){
      if(m>0){ return 1; }else {return 0;}
}
int foo(int x,int y){
      int result=0;
      if(x>0){  result=result+ bar(y); }
     else  if(y>0){result=result+ bar(x-y); }
      return result;
}
Summary Specification of bar(m):
input:{m0} ,output:{RET}, pre-condition: true,
path-set: {(m0>0, [RET=1]), (m0<=0, [RET=0])}
Summary Specification of foo(x,y):
input:{x0,y0}    ,output:{RET},    pre-condition:
true,
path-set: {(x0>0, [bar(y0), RET=bar(y0).RET]),
(x0<=0,[{(y0>0,[bar(x0-y0),RET=bar(x0-y0).RET]),
(y0<=0,[RET=0]) }])}
```

Fig. 1. Example 1 of summary specification

The loop in the code can be specified with recursive summary. For example, the loop code *while (i>0) { x=x-2; i--;}* can be specified as:

```
loop(i,x)={input:<i0,x0>, output:<i,x>, pre-condition: true,
    path-set:{(i0>0, [x= x0-2,i=i0-1 , loop(i0-1, x0-2)])}
```

2.3 Generation of Parameterized Summary

Figure 2 is an algorithm to generate the parameterized summary. With this algorithm, we can start from given method, and generate the summaries for methods in call graph and unbounded loops in methods.

The summary of given method or loop are generated through partially symbolic execution of given method. Instead of symbolic executing the whole application, we start symbolic execution from given method, and execute the codes in given scope. If we want to generate the summary for the method, the scope will be from the first code to the last code. Or if we want to generate a summary for one code fragment, e.g. a loop, the scope will be from the first code to the last code of the fragment. Line 6~40 of the algorithm forces the symbolic execution only executes the code within the scope.

Lines 7~16 cope with the tree-like path structure of summary. When the instruction is the first choice of the branch, we create an embedded path set step in the path, and set current path to the first path of this path set. When we explore the next choice, we create another path, and set current path to it. When the branch is completely explored, we set current path to its parent path.

Lines 17~27 cope with the data-written steps in the summary. If the instruction modifies a data that used after the scope, including the fields of *this* object, fields of argument object, static fields, or other variable used after the scope, we add a data-written steps to current path, and add the variable to the output set of the summary. Similarly, if the instruction reads data from a variable that defined before the scope, we add the variable to the input list of the summary. If the instruction is *return*, we add "ret" to the output set and recode the data-written step.

Lines 28~37 cope with the summary instance step of the summary. If instruction is the first code of an unbounded loop, or is an invoking instruction, we add a summary instance step to the path, create a summary generation task and put it into the task queue if it is required, set the output data to support the following symbolic execution, and skip the codes of the loop or invoking instruction.

This algorithm has good scalability. Since the algorithm is based on partially symbolic execution, the paths to be explored are localized into given scope of one method. Also, the unbounded loop and the invoked method are treated in separate summary generation tasks, and their paths will be explored only time. In addition, all the generation tasks have no dependencies with each other, so they can be scheduled in parallel.

```
Algorithm 1: generateSummary (M,start,end)

Input: method M and scope <start,end>
Output: summary S of  M in scope <start,end>
1  Path:=(true,[])  //current path;
2  S:=({},{},true, Path); //initiate the summary
3  Inst:= firstInstruction (M);
4  StartSymbolicExecution (M);
5  While(Inst!=null){
6    if(Inst.index>=start && Inst.index<=end) {
7       if(firstPCChoice(inst)){
8           newPath:=( LocalPC,[]);
9           Path.addPathSetStep({newPath});
10          Path:=newPath;
11      }else if(nextPCChoice(inst)){
12         newPath:=(currentLocalPC,[]);
13         Path.addSiblingPath(newPath);
14         Path:= newPath;
15      }else if(complatePCChoice(inst)){
16         Path:=parentof(P);
17      }else if(inst : storeInstruction){
18           if(scopeof(inst.var).end >end){
19              S.O= S.O∪{inst.var}
20              Path.addDataWrittenStep(inst.var, inst.value);
21           }
22      }else if(inst : loadInstruction){
23          if(scopeof(inst.varName).start < start )
24            S.I= S.I∪{(inst.varName, inst.value)} ;
25      }else if(inst: Return){
26           S.I= S.I∪{(ret, inst.value)}
27           Path.addDataWrittenStep(RET, inst.value);
28      }else if(inst.index = = Loop.start){
29         Path.addSummaryInstanceStep(Loop,Φ, PC);
30         enqueueTask (M, Loop.start, Loop.end);
31         setOutput(Loop.O);
32         skip(Loop);
33      }else If(inst: Invoke(M',ARG')&&isSymbolic(M')){
34         Path.addSummaryInstanceStep(M', ARG', PC);
35         If(!hasSummary(M') && ! inTaskQueue(M'))
36                enqueueTask (M', M'.start, M'.end);
37         setOutput(M'.O);
38         skip(inst);
39      }
40       SymbolicExecute(inst);
41   }// end of line 6
42   Inst := getNextInstruction(Inst);
43 }// end of line 5
```

Fig. 2. Algorithm of summary generation

3 Three Forms of Summary Instance

The summary we generated may have some summary instances in the form of abstract summary instance expression. However, when we use the summary in the test case generation or model checking, these summary instances need to be computed to get their concrete path set.

Based on the path set we computed, the summary instance has three forms.

3.1 Three forms of Summary Instance

Summary Instance has three forms: closed, expanded, and merged. As shown in Fig. 3, *Closed* is an abstract form that has an empty path set. *Expanded* form means that the path set of summary instance is computed and specified in multiple-path style. E.g. the summary instance *bar(y)* in expanded form will have path set that contains two paths: *{(y_0>0, [bar(y_0).ret=1]), (y_0<=0, [bar(y_0).ret=0])}*.

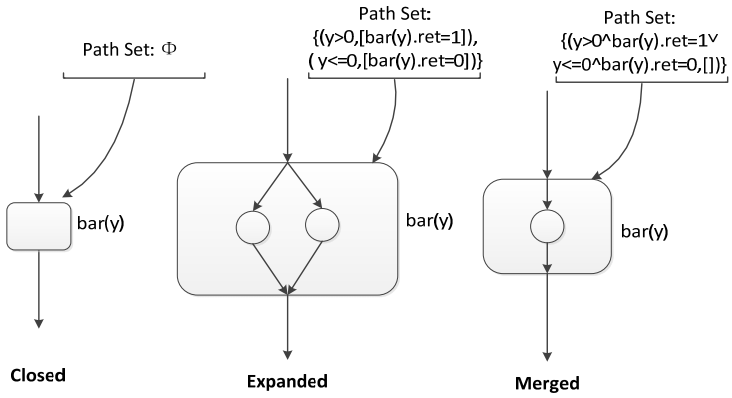

Fig. 3. Three forms of summary instance

Merged form means that all the paths of summary instance are merged into one path, which has a local path-condition same as the classical summary in [3]. In merged form, each path is translated into one path constraint firstly, which is a conjunction of path-condition with a proposition of the final state of the path. And that all paths are merged into a single formula with disjunction operators. E.g. the summary instance *bar(y_0)* in merged form will have a path set which contains one single path: *{(y_0>0^bar(y_0).ret=1∨ y_0<=0^bar(y_0).ret=0,[])}* .

Three forms of summary instance are able to meet different requirements of testing or verification. Closed form ignores the internal path set, and suitable for the situation that the path set is useless to the target. The expanded form has more paths to explore but simpler constraint to solve, while the merged form has more complex constraint but fewer paths. In the practice, we can trade-off between issues of path explosion and complex constraints solving by merging some of the summary instances while keep others expanded.

3.2 Path Expanding of Summary Instance

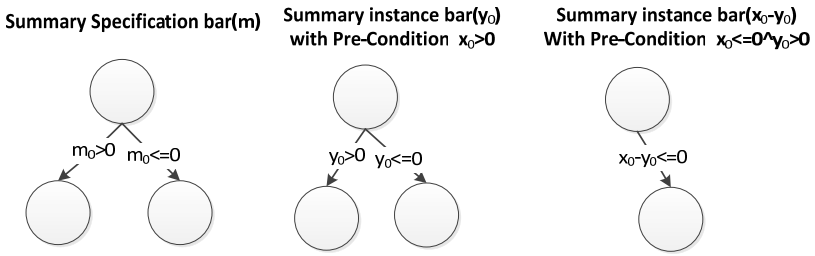

Fig. 4. Path expanding example of different summary instances

There are two tasks to expand the path set of summary instances: *substitution* and *pruning*. (1)Substitution: replace all the input variables in the paths in the summary specification with their corresponding symbolic expression in summary instance. (2) Pruning: removes the paths which conflict with the pre-condition in summary instance, and this task can be done by constraint solving. The algorithm of expanding is not complex, so it is not given in this paper. We simply explain it with an example.

In examples 1, the summary *foo(x,y)* has two summary instances *bar(y_0)* and *bar(x_0-y_0)*. bar(x_0) has a symbolic input value x_0, with a pre-condition x_0>0, and bar(x_0-y_0) has a symbolic input value x_0-y_0, with a pre-condition x_0<=0^y_0>0.

As shown in Fig. 1, the summary specification has two paths and one input variable *m*. when we compute the path set of *bar(y_0)*, we replace all the m_0 with *y,* and keep all paths since no path is conflict with pre-condition x_0>0, and finally, we get a path set *{(y_0>0, [bar(y_0).ret=1]), (y_0<=0, [bar(y_0).ret=0])}*. Similarly, in the case of

bar(x_0-y_0), we replace m_0 with x_0-y_0, remove the path x_0-y_0>0 because it conflicts with x_0<=0^y_0>0, and we get a path set *{(x_0-y_0<=0, [bar(x_0-y_0).ret=0])}*.

As to summary of loop, because it involves recursion, we should expand it for multiple times. If it is an unbounded loop, we should specify a number for the loop times just like what we do in symbolic execution.

3.3 Path Merging of Summary Instance

Path merging is a recursive process because of embedded path sets. Fig. 5 is a recursive algorithm which is able to merge a path set *PS* into a single path (PC,Φ).

In this algorithm, we build one path constraint for each path, and then compose them together with disjunction. Each path constraint is a conjunction of three parts: local path condition *Path.lpc*, constraint of sub-paths C_{sub}, and post condition.

Lines 2~16 travel the path to build the path constraint. Lines 7~12 show that if the step is an embedded path set or an expanded summary instance, we merge its path set together into one path, and compose it into $C_{sub.}$ Lines 13~14 show that if the step is a data-written step, we put the value and its symbolic value into MAP. After the path traveling (in Line 17), the path constraint is built as a conjunction of local path condition, constraint of sub-paths, and post condition. we build the post condition with MAP, which recodes the final value of output data.

Line 18 composes the path constraints together with disjunction. When all the path constraints are generated and composed, we build a merged path with the final path condition *PC*, and an empty trace of path.

The complexity of algorithm is O(n), because we travel every path (including the sub-paths) only once in the algorithm.

```
Algorithm 2: merge (PS)
Input: target path set PS
Output: merged path (PC,Φ)
1  PC=null; //the final path condition of merged path
2  for(each Path ∈ PS) {
3      for(each step ∈ Path.T) {
4        Csub=true; // constraint for sub-paths
5        MAP=Φ;//hash map to store data states of path
6        if(step is a summaryInstance){
7            if(isExpanded(step)){
8                P':=merge(step.PS);
9                Csub':=  Csub' ^ P'.PC }
10        }else if(step is a path-set){
11             P':=merge(step);
12             Csub':=  Csub' ^ P'.PC;
13        } else if(step assign value to var){
14             MAP.put(var,value);
15        }
16      }////end of  line 2
17      PC' := Path.lpc  ^ Csub ^ PostCondition(MAP);
18      PC  := PC ∨ PC';
19   }//end of  line 1
20    Return (PC,Φ);
```

Fig. 5. Algorithm of summary generation

4 On-demand Path Expanding and Merging of Parameterized Summary

Based on parameterized Summary, we propose an approach called DEMPS (on-Demand Expanding and Merging of Parameterized Summary). This approach is able to expand and merge the paths in summary according to the target of testing or verification, as well as the requirement of scalability.

4.1 What is On-demand Path Expanding of Summary?

When we use parameterized summary to support test case generation or model checking, it is possible that some of the summaries instance are useless for the target. E.g. If we want to generate the test cases that cover all paths

of example *foo(x,y)*, the two instances $bar(x_0)$ and $bar(x_0\text{-}y_0)$ are all useless, because they are not used in the path-conditions. However, if we want to verify the post-condition properties "ret>0" for foo(x,y), this two instances are useful, because the value of *ret* is related to them.

Taking this fact into consideration, one of the best practices to use the summary is that we should expand the summary instances that contribute to the target, and keep the useless summary instances closed. We call this *on-demand summary expanding*. In this way, we can get fewer paths to explore as well as simpler constraints to solving.

4.2 What is On-demand Path Merging of Summary?

Summary instances in merged form have the advantage in reducing the amount of path exploration, but similar to the traditional summary, they have the negative effects on the constraint solving. On the other hand, the expanded form will result in simpler constraint than merged form, but it will have more paths to explore. If the constraints are too complex or the paths amount increase exponentially, the cost of symbolic execution will be very expensive.

Therefore, we should have a trade-off to keep a balance between path exploration and constraint complexity. In our approach, we can selectively merge given set of the summary instances, while keep others expanded to achieve this balance. This is so called *on-demand summary merging*.

4.3 Algorithm of On-demand Summary Expanding and Merging

Figure 6 gives a general algorithm for on-demand summary expanding and merging. This algorithm has three inputs, where *P* is the target path to be tested or verified, *D* is the target data set used in test path or checked properties, and M is the summary instance set to be merged. This algorithm travels the path set from target path to the root in bottom-up order to find the summary instances to be expanded and merged.

```
 Algorithm 3: ondemandExpandAndMerge (P, D,M)
 Input: target path P of given summary, target data set D,
Summary instance set to be merged
 Output: the summary whose  path set is expanded
   1 CP:=P; // current path
   2 SI:={};//summary instances to be expanded
   3 while(CP!=null){
   4      //backward travel the trace
   5     for(each step ∈ CP.T from the last to the first) {
   6       if(step is summaryInstance&& step.O∩D !=Φ){
   7           SI:=SI∪{step};
   8        }else if(step assign value to var && var∈ D){
   9           D := D∪summaryOutputUsedIn(value);
   10       }
   11    }//end of  line 4
   12    D := D∪SummaryOutputUsedIn(CP.lpc);
   13    CP:=parentof(CP);// bottom-up traveling
   14  }//end of  line 3
   15for(each si ∈ SI){
   16    if(isClosed(si)){
   17        PS := computePathSet (si);
   18        for(P' ∈ leaviesOf(PS)){
   19           ondemandExpandAndMerge (si.S, P', D∩si.O);
   20        }
   21   }//end of line 16{
   22   If(si ∈ M && isExpanded(si) ) {
   23        mps := merge(si .PS);  si.PS={ mps }; }
   24} //end of line 15
```

Fig. 6. Algorithm of summary generation

The algorithm visits the paths in a bottom-up and backward order, because the data only depend on its former steps. It starts form the target path, and travels the path backward, after the path is traveled, then go on traveling its parent path.

Lines 3~14 do the bottom-up traveling, where line 13 turns to its parent path. Lines 5~11 do the backward traveling for current path. Lines 6~7 indicate that if current step is a summary instance which has outputs in target data set D, then we add this summary instance into SI, a set of summary instance to be expanded. Lines 8-9 indicate that if current step is a data-written step, and the modified variable is in target data set D, we check the value expression of this variable, and if it uses any summary instance's output, we put these summary instance's outputs into data set D. After traveling current path, we check the current local path condition, and put the summary instance's output it used into data set D.

Lines 15~24 do the expanding and merging for the summary instances in SI. Lines 15~20 expand the summary instances reclusively. It computes the path set of summary instance and on-demand expands and merges all of its leaf nodes with a target dataset D∩si.O. Lines 22~23 check if the summary instance is in merged set M. If so, then merge it.

5 Application of Parameterized Summary

Parameterized summary supports test case generation and model checking in two different ways: (1) symbolic execution; (2) without additional symbolic execution.

5.1 Using Parameterized Summary with Symbolic Execution

Traditional summaries need not standalone generation procedure, and are generated and used in the process of test case generation. Each summary is specified as one formula which merges the pre and post conditions of paths of target methods together. If the symbolic execution invokes the method again in the same context, it can be reused and without re-execute the target method again.

In our approach, the summary instance is the counterpart concept to the traditional summary. The merged form of summary instance has a path condition that is as same as the traditional summary. Therefore, at least, the summary instance can be used as a traditional summary to support the symbolic execution. Furthermore, we provide three forms of summary instances which can be used to meet different requirement.

1) Lazily expanding and merging of summary instances

Closed form of summary instance can be used to support lazily expanding and merging. When the symbolic execution encounters a method invoking, we get the summary specification we generated before, and create a summary instance with symbolic values of arguments as its input and current path condition as its constraint. And then set the abstract expression of return value or other output values to the state of symbolic execution. But will not change the current path condition immediately.

We will keep this summary closed until their output expressions appear in the constraints to solve. E.g. in the example of Fig. 1, if we want to generate the test cases for the *foo* method, the summary instances bar(y_0) and bar(x_0-y_0) will keep closed all the time, because we never use them in the path conditions.

However, if we verify the post condition property "ret>0" through constraint solving. In the path *if(x_0>0){result=result+ bar(y_0);}*, the negative constraint to solve is *ret<=0 ^ x_0>0 ^ ret=bar(y_0).ret*, which uses the output of one summary instance *bar(y).ret*. So we will expand and merge paths of *bar(y_0)*, and compose the merged path condition with the original constraint, then we get a constraint to solve: *ret<=0 ^ x_0>0 ^ ret=bar(y_0).ret ^ (y_0>0^bar(y_0).ret=1 ∨ y_0<=0 ^ bar(y_0).ret=0*). By solving this constraint, we will get a count example. Similar we should expand and merge bar(x_0-y_0) to check the property for another path.

2) Usage of Merged and Expanded Summary Instance

We can use both expanded and merged form of summary instance in symbolic execution. The merged from can be used like traditional summary. When the path condition contains the summary instance expression, we add the merged path condition to current path condition to support the path solving.

The expanded summary instance will be a little different. Because it contains multiple paths, we should treat is just like the branch instruction in symbolic execution, and create a group of sub-paths in symbolic tree. E.g. if we use the expanded form of bar(y_0), we will create a two-path choices in symbolic tree. One path uses *y_0>0^bar(y_0).ret=1*as its local path condition, and another path uses *y_0<=0^bar(y_0).ret=0* as its local path condition.

5.2 Using Parameterized Summary without Additional Symbolic Execution

In other view, the summary is the behavior specification of the program, and it includes information to support test case generation and verification. Therefore, based on them, we can do some works without additional symbolic execution.

In this section, we show an example of on-demand test case generation to illustrate how to use the summary without symbolic execution.

1) Problem analysis

On-demand test case generation in this paper means that we generate the test case for specific path or specific line of code. Generating such test cases based on parameterized summary means that we need to find a group of inputs with which the program will go along a path to the target node in summary paths. Therefore, on-demand test case generation is reduced to one problem: *finding one feasible path led to the target node in the summary paths.* Once we find this path, the input values that solved in this path will be the test case we need.

The example code as well as its summary specifications is shown in Fig. 7. In this example, we want to generate the test case for the code *if(sum ==0) assert(false)*, and the corresponding path in summary is *(bar(s1).ret+ bar(s2).ret+ …+ bar(s19).ret ==0,[ERR]).*

The code of this example is quite simple, but traditional symbolic execution doesn't work well to generate the test cases, because there are about 2^{51} paths to be explored, caused by the method invoking in the loop. Traditional summary-based method only explores three paths, but has to solve three complex constraints, each of which contains 201 propositions, therefore will also cost a lot of time.

Example Code 2:

```
static int N = 50;
int bar (int m) {
    if(m>0) return 1; else  return -
1;
}
int top(int [] s) {
    int sum = 0;
    for (int i=0; i< N; i++) {
       sum = sum +bar(s[i]);
   }
   if(sum ==0) assert(false);
   return bar(sum);
}
```

Summary Specification of bar:

```
input:<m>    ,output:{ret},    pre-
condition: true,
path-set:  {(m>0,  [ret=1]),  (m<=0,
[ret=-1])}
```

Summary Specification of top:

```
input:<s>   ,output:{ret   },   pre-
condition: true,
 path-set:  {(true,[bar(s0),bar(s1),
 bar(s2)  …bar(s19),  {(bar(s1).ret+
 bar(s2).ret+…+bar(s19).ret==0,
[ERR]),
(bar(s1).ret+         bar(s2).ret+…+
bar(s49).ret !=0,
[bar(bar(s1).ret+
bar(s2).ret…+bar(s19).ret)),
ret=
bar(bar(s1).ret+…+bar(s49).ret)).re
t])})}
```

Fig. 7. Algorithm of summary generation

2) Steps of our approach

Our approach has two steps to generate the test case based on parameterized summary. In the first step, we expand and merge the summary according to the target, in order to reduce the complexity and get balance between path exploration and constraint solving. In the second step, we search the paths according to specific strategy, and

solve the constraints composed by the path conditions along the paths. Once we get a feasible solution in one path, the input values in the solution will be the test case we need.

Step1: on-demand expanding and merging of summary

Base on the method in section IV, we can expand and merge the summary on demand. In this example, the target path is *(bar(s1).ret+ bar(s2).ret+…+ bar(s49).ret ==0,[ERR]),* which constraints 50 summary instance outputs {bar(s0),bar(s1),…bar(s49)}, therefore, we should expand or merge the summary instance bar(s0),bar(s1),…bar(s49). As shown in Fig.7, we decide to merge 12 summary instances and keep 8 instances expanded. Each of the merged summary instances has only one internal path, and each of the expanded summary instances has two internal paths, therefore, there will be totally 2^{25} paths form the first node to the target.

The summary *bar(bar(s1).ret+ bar(s2).ret +…+bar(s49).ret)* is not used directly or indirectly by the target path, therefore, we keep it closed.

a) **Step2: path searching and constraint solving**

After step 1, we get a group of paths. Fortunately, we need not to solve constraints for all paths, because our target is to find one feasible path. Once find it, the searching and solving will stop immediately. In the example, the 2^{25} paths generated are all feasible. Therefore the searching and solving will stop after the first path.

The path searching can be in depth-first order, breadth-first order, random order, or order in heuristic strategy. We would better adopt the random or heuristic search strategy, because in general they find the feasible path more quickly.

Whatever the search strategy we adopt, once the search gets a path from root to the target node, we compose together all the path conditions of all nodes along the paths into a path constraint to solve. In the example, if the search get a path which chooses the first internal path for all the expanded instances *bar(s12),bar(s13)…bar(s49)*. Then we can compose together the path conditions of *bar(s0), bar(s1)…,bar(s24)* with the path conditions *s25>0, s26>0,…s49>0*, and the path condition *bar(s1).ret+bar(s2).ret+…bar(s49).ret==0*. The composed path constraint is:

```
(s0>0^bar(s0).ret=1∨s0<=0^bar(s0).ret=-1) ^
(s1>0^bar(s1).ret=1∨s1<=1^bar(s1).ret=-1) ^
…
(s11>0^bar(s11).ret=1∨ s11<=0^ bar(s11).ret= -1) ^
S25>0^ s26>0^ …  s49>0  ^
bar(s1).ret+bar(s2).ret+…bar(s24).ret+25==0
```

This constraint has 126 propositions, and is much simpler than the constraints generated by traditional summary method.

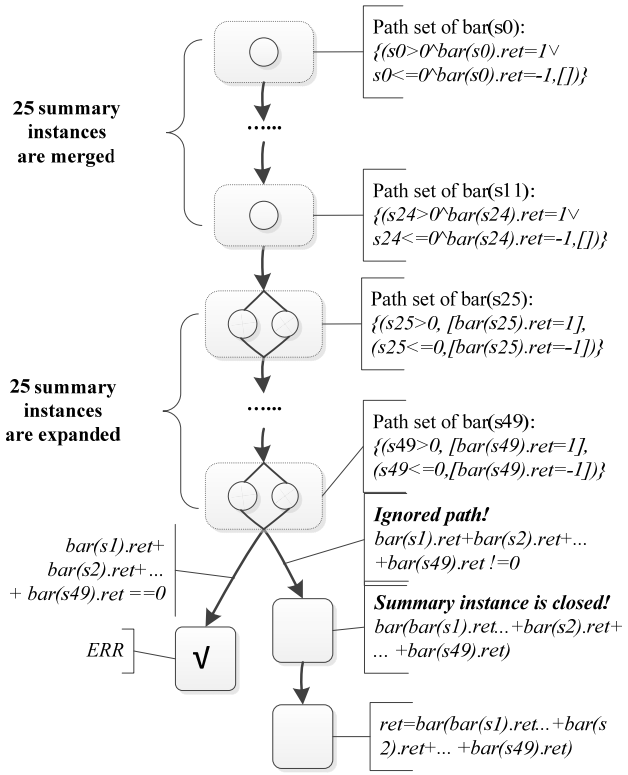

Fig. 8. Example of on-demand path expanding and merging

We solve this path constraint. If the constraint is satisfied, then this is a feasible path. We get the input values from the resolution of this constraint, and then test case is generated by now. If the constraint is not satisfied, we continue and get the next path to solve until we find a resolution. If all the paths' constraints are solved but none is satisfied, that means the target path is unreachable in the program and has no test case.

The constraints generated by search can be solved in parallel, because the constraint solving tasks have no direct dependency with each other. The search generates multiple paths, and puts into parallel scheduler, which arranges the path constraints in parallel with the help of multicore or distributed computing techniques.

6 Experiment and Evaluation

6.1 Implementation of DEMPS approach

We implement DEMPS approach based on Symbolic JPF [4]. Symbolic JPF is an expansion of java model checking tool JPF (Java Path Finder). It provides the implementation for efficient dynamic symbolic execution, and supports test case generation and bug finding, as well as assert verification. We implement DEMPS approach as a JPF plugin. The main classes we implemented include:

(1) PartialExecutionVM: an extension class of SingleProcessVM. It enables partial symbolic execution that can start the symbolic from any method, instead of just the static main method.

(2) SummaryGenerator: a listener of JPF, which generates the summary for the methods or the loops, by responding to the events of symbolic execution.

(3) SummaryGenerationTaskManager: maintains a task queue of summary generation, and starts a partial symbolic execution to enact the task.

(4) SummaryDAOService: access to a Redis database in order to store and read summaries and summary instances's path set.

(5) SummaryAssembler: implement the path expanding and merging algorithms, as well as the algorithm for on-demand expanding and merging.

(6) RandomSearch: search in random order in the paths of summary, in order to find the paths led to the target.

(7) ConstraintSolvingTaskManager: a parallel task scheduler for constraints solving.

Also, we modified the class PathCondition to support adding a disjunction expression to the constraint, and translate it into CNF style so that it can be handled by Coral solver.

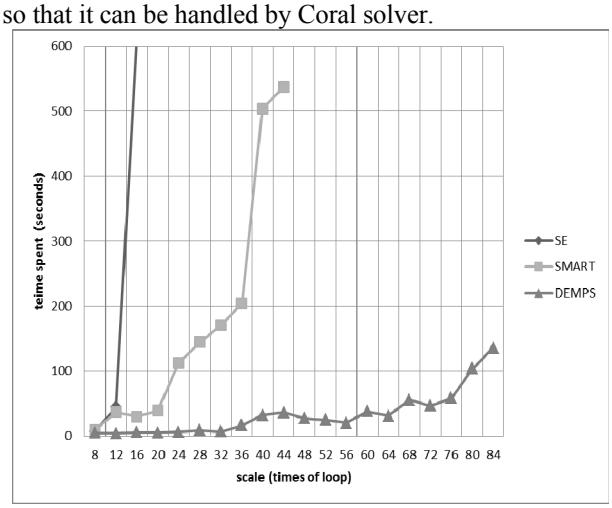

Fig. 9. Experiment result of scalability of three approaches

6.2 Research Questions

In this section, we want to answer two research questions through experimental evaluation.

1) Will parameterized summary improve the scalability of test case generation or verification?

This is the main benefit of parameterized summary approach that we advocate. Does parameterized summary perform better than traditional summary and non-summary symbolic execution? We will evaluate the performance of these three approaches with different scales of programs.

2) What are the performance effects of summary merging and expanding?

In order to get better performance, should we merge more summary instances or expand more? Which summary instances should be merged and which ones should be expanded? We will evaluate the performance of our approach in different situations to answer these questions. The answer will give a guide on the trade-off between path exploration and constraint solving.

6.3 RQ1-Improvement of Scalability

We did an experiment in symbolic JPF, to evaluate the scalability of our approach, by comparing DEMPS approach with (1) symbolic execution without summary (written as SE), and (2) SMART approach, which is based on classical pre/post-condition based summary.

Symbolic JPF supports test case generation through dynamic symbolic execution, but does not provide the implementation of SMART approach. We contacted with the author Patrice Godefroid, who replied that his group had no implementation of SMART in JPF. Therefore, we implemented a prototype of SMART method by ourselves. This prototype is a little different from SMART: we generate the summary with full paths of method, instead of partial paths in respect to the invoking context in SMART. But this won't influence the result of our evaluation, because all the invoked method in this experiment has no deference on this point.

We choose the example 2 in Section V as the target program to be processed. This example has a method invoking in the loop, and will generate huge number of paths in symbolic execution, which will be a challenge for both SE and SMART methods.

This experiment generates the test cases with three different approaches, for code line *if(sum ==0) assert(false)* in programs of different scales . We assign the loop times N to different values, to generate different scales of symbolic execution. In the experiment, we changed the loop times N from 8 to 84, and the paths of symbolic execution will scales from 2^9 to 2^{85}.

In the experiment, we use Coral [23] as the constraint solver, because it handles disjunction well in JPF. We set "multiple errors" to *false* to force the symbolic execution stop as soon as it finds the error. Also, we generate summary specifications for method *foo* and *bar* before evaluation (time spent is less than 2 seconds). We expand the summaries according the target code, and merge half of the summary instances of *bar*(y), and keep another half expanded.

And after the test case generation with three approaches in different scales, we get a result as shown in fig. 9. In this figure, the X-axis is the times of loop, which indicate the scale of symbolic execution. The Y-axis is the time spent on finding the test case.

It is clearly shown that symbolic execution without summary has poor scalability. When N increases to 16, the time cost will be unacceptable.

The SMART method has a better scalability than SE. The reason is that: no matter how many the times of loop is, it only has two paths to explore, and only has to solve constraints twice in each symbolic execution. However, when loops number increases more than 40, the time cost increases dramatically, because it spends a lot of time on solving the complex constraints, and each of which has 4N+1 propositions.

As shown in the figure, the DEMPS approach has a better scalability than other two approaches when we merge half of the summary instances while keep another half expanded. The maximum loop times can reach more than 80. In this experiment, our approach generates $2^{N/2}$ paths, but these paths are all feasible. Therefore only one time of constraint solving is required to generate the test case. Each constraint contains 2.5N+1 proposition, which is much simpler than the constraints in SMART approach.

6.4 RQ2- Performance Effects of Merging and Expanding

The performance of DEMPS approach largely depends on the decision of summary expanding and merging. Though we get a good result in experiment 1, but if we choose an unsuitable decision, it is possible that we might get a worse result than the classical method. So there must be a trade-off between path exploration and constraint solving to get better performance.

We design an experiment to find out what will influence the performance of our method, and what is the best practice to decide which summary instances should be merged and which ones should be expanded.

The program to test is a modified version of example in Section V. the program has another method foo2, with a code like this:

```
void  foo2(int[] A, int[] B)
 int sum1 = 0; int sum2=0;
 for (int i = 0; i <20; i++) {
     sum1 = sum1 + bar(A[i]);sum2=sum2+bar(B[i]);
 }
```

```
  if(sum1>=0 && sum2>12) { assert (false);}
}
```

The target is to generate the test case for line *if(sum1>=0 && sum2>10){assert(false);}*. Firstly, we generate the summary for *foo2*. As a result the target path condition is: *bar(A1).ret+bar(A2).ret+...bar(A19).ret >=0 ^ bar(B1).ret+ bar(B2).ret+... bar(B19).ret>= 12*. This path condition can be divided into two parts. The first part *bar(A1).ret+ bar(A2).ret+...bar(A19).ret>=0* is easy to solve because there are a lot of feasible paths in symbolic tree. The second part *bar(B1).ret+bar(B2).ret + ...bar(B19).ret>=12* is difficult to solve because there are just a few feasible paths in symbolic tree.

In other words, summary instances *bar(B1)~bar(B19)* have more possibility to cause infeasible paths than *bar(A1)~bar(A19)*. In this experiment, We tried different decisions of summary merging and expanding, to find out what is difference between merging/expanding *bar(A1)~bar(A19)* and *bar(B1)~bar(B19)*.

The experiment result is shown in TABLE 1. The table head and the first column is the merging scope of two groups of summary instances. Each cell of the table contains two figures. The first figure is the time spent to generate the test case, and the second one is the count of searched path. (51.469, 8) means it searched 8 paths and the total time cost is 51.469 seconds. *Timeout* in the cell means it spent more than 300 seconds to find the test case.

As shown in the table, it seems that merging *bar(B1)~bar(B19)* has positive effect on the performance, while merging *bar(A1)~bar(A19)* has negative effect on the performance.

This can be explained in respect to the effect of merging on paths. In DEMPS approach, the path constraints in summary instances *bar(B1)~bar(B19)* will be composed with the target path constraint *bar(B1).ret+bar(B2).ret + ...bar(B19).ret>=12*. However, among these composed constraints, a great deal of them is infeasible. Merging several paths into one has a merit to eliminate some infeasible compositions, because the disjunction of several path constraints provides more choices to compose with sequential path constraints, so that lead to higher possibility of feasible composition.

However, there is another interesting phenomenon in the table: the best performance appears in row bar(B0) ~bar(B14) instead of row bar(B0)~bar(B19). The reason is that: after merging bar(B0) ~bar(B14), the possibility of feasible paths is already 100%, but it has simpler constraint than that of merging bar(B0)~bar(B19).

Table 1. Experiment result of effects of merging and expanding

Merging Scope	none	bar(A0) ~ bar(A4)	bar(A0) ~ bar(A9)	bar(A0) ~ bar(A14)	bar(A0) ~ bar(A19)
none	timeout	timeout	timeout	timeout	timeout
bar(B0) ~ bar(B4)	136.235,21	timeout	timeout	timeout	timeout
bar(B0) ~ bar(B9)	51.469,8	timeout	timeout	timeout	timeout
bar(B0) ~ bar(B14)	10.749,1	14.690,1	85.571,1	154.688,1	timeout
bar(B0)~ bar(B19)	11.877,1	36.060,1	41.038,1	timeout	timeout

Therefore, we can make a conclusion: merging summary instances are able to improve the possibility of feasible path composition, as well as decrease the paths to explore; while expanding is able to decrease the complexity of constraints. Therefore, if the summary instance has high possibility to cause infeasible paths, merging will have positive effects on the performance, and if the summary instance has low possibility to cause infeasible paths, expanding will have positive effects on the performance.

In the practice, we can order the summary instances according their possibility to cause infeasible paths, and merge the summary instances which have high possibility, until it produces a good enough possibility of feasible path. One of the challenge problems is how to precisely estimate the possibility that cause infeasible paths. This will be one of our future works.

7 Related Work

Compositional symbolic execution or summary-base method has been developed since Patrice Godefroid proposed it in 2007[20]. We will list and discuss some important works that related to ours.

Incomplete summary: *Incomplete summary* means that the summary does not cover the full paths of target method [21,22]. In [21], summary contains dangling nodes that represent unexplored paths, and are able to be expanded lazily on a demand-driven basis. Therefore as few intra-procedural paths as possible are symbolically executed in order to form an inter-procedural path leading to a specific target branch or statement of interest.

In our approach, summary specification is regarded as a complete summary in the view of intro-procedural analysis, while summary instances are incomplete one. Similar to [21], the summary instance in our approach is also able to be expanded on demand. We distinguish the summary specification and summary instances, in order

to get better reusability and flexibility: the summary specifications and summary instances are all reusable, and summary instances can provide different degrees of completeness to meet different requirements.

Abstract summary: *Uninterpreted functions* is allowed to appear in summaries [20,21,22,24]. Such summary is called abstract summary in [22]. In general, uninterpreted functions are utilized to express the functions that are difficult or impossible to be symbolically executed. When the summary with uninterpreted functions merge into path constraint, such path constraint is possible to be solved with mixed concrete-symbolic solving in dynamic symbolic execution, or be utilized to support high-order test case generation [24].

In our approach, the summary which contains summary instances is somehow like the abstract summary, because the summary instance expression has similar characteristics as that of uninterpreted function. However, the main feature of the summary instance expression is to support the summaries composition, as well as on-demand expanding and merging, instead of to represent an "unknown" function.

Loop Summary: Input-dependent loops may cause an explosion in the number of constraints to be solved and in the number of execution paths to be explored. Loop summary is able to simplify the execution of loop into one first logic formula, which is the conjunction of pre and post condition of the loop[25,26]. In [25], Patrice Godefroid proposed an algorithm to generate the loop summary. The algorithm includes a (partial) loop-invariant generator that uses pattern-matching rules on the loop guards to guess the number of loop iterations, and can infer loop invariants relating values of induction variables, a restricted but common class of loop invariants.

Parameterized summary is also helpful to reason the pre and post-condition of loop summary. For instance, in the loop example in Section II, where the summary path set is: *{(i_0>0, [x=x_0-2,i=i_0-1, loop(i_0-1, x_0-2)])}, g*iven x and i has the initial symbolic value x_0 and i_0. Obviously, the pre-condition of loop is $i_0>0$. With the help of this summary, we can reason that after *N* times of loop, the variables: $i=i_0-N$ and $x=x_0-2*N$. When the loop is completed, $i_0-N==0$, therefore the post condition of loop will be $i==0 \wedge x==x_0-2*i_0$.

Path/state merging: Path merging is an approach to decrease the path number to be explored through merging them using select expressions [14,15,16]. E.g. two paths (X < 0, [x = 0]) and (X ≥ 0, [x = 5]), can be merged into one path (true, [x = ite(X < 0,0,5)]). Similar to the summary-base method, state merging also has the problem that it will increase the complexity of constraint solving. Volodymyr Kuznetsov presents query count estimation, a method to automatically choose when and how to merge states in order to get better performance [16]. The method statically estimate the impact that each symbolic variable has on solver queries that follow a potential merge point; states are then merged only when doing so promises to be advantageous.

In our approach, it is also a problem to decide when and how to merge paths of summary instance. The method of Volodymyr Kuznetsov mainly considers the merging's effects on complexity. However, our experiment shows that the effect on the path feasibility is also an important factor to be taken into consideration, because the time cost on solving unfeasible constraints is very expensive.

Precision and progress of compositional symbolic execution: Dries Vanoverberghe and Frank Piessens point out in [28] that precision and progress are two important properties hold by classical symbolic execution, but difficult to be proved in compositional symbolic execution. They give a formal definition of precision and progress, as well as an algorithm to check them for compositional symbolic execution.

According to this paper, DEMPS approach is precise, because for every invocation in the application, all the leaf nodes of the summary instances are reachable in the calling context, and the return value of summary doesn't introduce any approximants. DEMPS approach doesn't satisfy the strong process property, because it adopts on-demand expanding which introduces unfairness. But DEMPS approach is complete relatively for test case generation or bug finding, because all the internal paths of called method that lead to the target are kept for exploration.

8 Conclusion

In this paper, we proposed an approach named DEMPS, to improve the scalability of compositional symbolic execution. The main contributions of our work include:

(1)We presented parameterized summary, which is able to be instantiated by calling context, as well as be composed in hierarchical way. Also, an algorithm based on partially symbolic execution is given to generate the summaries.

(2)We defined three forms of summary instance: closed, expanded, and merged to meet different requirements of test case generation or verification.

(3)We proposed an approach DEMPS based on parameterized summary, which is able to improve the scalability through on-demand path expanding and merging of summary.

(4)We implement the DEMPS approach in symbolic JPF, and did the experimental evaluation for the approach. And the experiments show that if we choose suitable decision of expanding and merging, the DEMPS will get better scalability than classical symbolic execution and classical summary-based method.

In the future, we would like to do further work on evaluating the effect of path merging and expanding, in order to get a more precise trade-off between exploration and constraint solving. We will also do some works on incremental approach for compositional symbolic execution [27,29] based on parameterized summary. In addition, we would like to analyze mobile apps [30] or identify malicious codes [31] for large programs based on our approach.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 61272108, No.539034.

References

[1] J.C. King, "Symbolic execution and program testing," *Communication of the ACM*, Vol. 19, No. 7, pp. 385-394, 1976.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference*, ACM Press, pp. 263-272, 2005.

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

[4] C.S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, Vol. 20, No. 3, pp. 391-425, 2013.

[5] C. Cadar, K. Sen, "Symbolic execution for software testing," *Communication of the ACM*, Vol. 56, No. 2, pp. 82-90, 2013.

[6] C. Păsăreanu, W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, No. 4, pp. 339-353, 2009.

[7] S. Anand, *Techniques to Facilitate Symbolic Execution of Real-World Programs*, Ph.D. Dissertation, Georgia Institute of Technology, 2012.

[8] J. Burnim, K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443-446, 2008.

[9] R. Majumdar, K. Sen, "Hybrid concolic testing," in *Proceedings of International Conference on Software Engineering*, pp. 416-426, 2007.

[10] J.H. Siddiqui, S. Khurshid,"Scaling symbolic execution using ranged analysis," in *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 523-536, 2012.

[11] Y. Li, Z. Su, L. Wang, X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 19-32, 2013.

[12] P. Boonstoppel, C. Cadar, D. Engler,"RWset: Attacking path explosion in constraint-based test generation," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 351-366, 2008.

[13] R. Majumdar, K. Sen, *Latest: Lazy Dynamic Test Input Generation*, Tech. Rep. UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007

[14] T. Hansen, P. Schachte, and H. Sondergaard, "State joining and splitting for the symbolic execution of binaries," in *Proceedings of International Conference on Runtime Verification (RV)*, pp. 76-92, 2009.

[15] P. Collingbourne, et al., "Symbolic crosschecking of floating-point and SIMD code," in *Proceedings of the Sixth Conference on Computer Systems*, ACM Press, pp. 315-328, 2011.

[16] V. Kuznetsov, J. Kinder, S. Bucur, G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 193-204, 2012.

[17] K. Taneja, T. Xie, N. Tillmann, J. de Halleux,"eXpress: guided path exploration for efficient regression test generation," in *Proceedings of International Symposium on Software Testing And Analysis*, pp. 1-11, 2011.

[18] S. Person, G. Yang, N. Rungta, and S. Khurshid,"Directed incremental symbolic execution," in *Proceedings of ACM SIGPLAN Conference On Programming Language Design And Implementation*, pp. 504-515, 2011.

[19] T. Avgerinos, A. Rebert, S.K. Cha, and D. Brumley,"Enhancing Symbolic Execution with Veritesting," in *Proceedings of 36th International Conference on Software Engineering*, pp. 1083-1094, 2014.

[20] P. Godefroid,"Compositional dynamic test generation," in *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47-54, 2007.

[21] S. Anand, P. Godefroid, N. Tillmann,"Demand-driven compositional symbolic execution," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367-381, 2008.

[22] S. Person, M.B. Dwyer, S. Elbaum, C.S. Păsăreanu, "Differential symbolic execution," in *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 226-237, 2008.

[23] M. Souza, M. Borges, M. D Amorim, C.S. Păsăreanu, "Coral: Solving complex constraints for symbolic pathfinder," in *Proceedings of NFM 2011*, Volume 6617 of Lecture Notes in Computer Science, pp. 359-374, 2011.

[24] P. Godefroid, "Higher-order test generation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design And Implementation*, pp. 258-269, 2011.

[25] P. Godefroid, D. Luchaup,"Automatic partial loop summarization in dynamic test generation," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 23-33, 2011.

[26] J. Strej Ek, T. Marek, "Abstracting Path Conditions," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 155-165, 2011.

[27] C.-Y. Chen, V.-W. Soo, "The step similarity comparisons on method patents," *Journal of Internet Technology*, Vol. 9 No. 4, pp. 393-401, 2008.

[28] D. Vanoverberghe, F. Piessens,"Theoretical aspects of compositional symbolic execution," in *Proceedings of Fundamental Approaches to Software Engineering*, pp. 247-261, 2011.

[29] P. Godefroid, S.K. Lahiri, C. Rubio-González, "Statically validating must summaries for incremental compositional dynamic test generation," in *Proceedings of the 18th International Conference on Static Analysis*, pp. 112-128, 2011.

[30] S.-h. Ju, H.-s. Seo, J. Kwak, "Study on analysis methodology for android applications," *Journal of Internet Technology*, Vol. 14, No. 5, pp. 851-858, 2013

[31] D.-C. Huang, H.-C. Lo, P.-L. Lai, W.-M. Chen, "A multiple pattern matching method for malicious code detection," *Journal of Internet Technology*, Vol. 13, No. 2, pp. 181-194, 2012.