Compositional Symbolic Execution through Program Specialization

Jose Miguel Rojas[a], Corina S. Păsăreanu[b]

[a] *Technical University of Madrid, Spain*
[b] *CMU-SV, NASA Ames, Moffett Field, CA, USA*

Abstract

Scalability is a major challenge in symbolic execution. The large number of paths that need to be explored and the large size of the constraints that must be carried often compromise the effectiveness of symbolic execution for software testing in practice. Compositional symbolic execution aims to alleviate these scalability issues by executing the methods of a program separately, stowing their results in method summaries and using such summaries to incrementally execute the complete program. We present a novel compositional approach that leverages partial evaluation, a well-established technique that aims at automatically specializing a program with respect to some of its input. We report on its design and implementation in Symbolic PathFinder and on preliminary promising evaluation results.

Keywords: Compositionality, Program specialization, Symbolic execution

1. Introduction

Scalability is a major challenge in symbolic execution [12, 4]. The large number of paths to explore and constraints to solve often hinders the applicability of symbolic execution for software testing in practice. Compositional reasoning is a general purpose methodology that has been used with success in the past to scale up static analysis and software verification techniques. The main idea is to analyze each elementary unit (methods or procedures) in the program separately, stowing the results in procedure *summaries* typically represented as logical constraints. Whole-program results are obtained by incrementally composing and re-utilizing the summaries obtained for each of its parts.

Recently, compositional reasoning has been considered in the context of symbolic execution and test case generation [2, 7, 1]. Although quite effective, the technique can become expensive in the presence of program operations that update the heap. These operations can not be easily encoded and checked as logical constraints, as in the summaries proposed in [2, 7], where they are not treated. On the other hand, the work in [1] explicitly encodes the input and output heap in the procedure summaries. However, this entails a complicated compositional

operator, which not only checks compatibility at invocation point, as all compositional techniques do, but also needs to synthesize the new program state (with the new heap) to continue execution.

We propose an alternative approach to compositional symbolic execution in the presence of heap updates, which is based on *partial evaluation (PE)* [8]. PE, also known as *program specialization*, is a well known technique for automatically specializing a program with respect to some of its inputs. In our approach a method summary consists of a set of summary cases, corresponding to all the symbolic paths through the method; each summary case contains the path condition and heap constraints that enable a particular symbolic execution path, together with a "path-specialized" version of the method code, obtained through PE. Thus the operation of composing a method summary with the actual calling context consists of re-executing a summary case according to the specialized code which naturally reconstructs the heap without the need of keeping an explicit representation in the summary.

Using program specialization techniques helps optimize the symbolic execution process by enabling preemptive pruning of unfeasible branches and allowing deeper exploration of the symbolic execution state space. Last but not least, a byproduct of our program specialization-based compositional approach is that the code stored in each method summary case can be further used as a specialized, more efficient version of the original method.

We have implemented our compositional analysis in Symbolic PathFinder [11] for the symbolic execution of Java bytecode programs. We report on preliminary encouraging results on two small case studies; more experimental evaluation is planned for future work.

More Related Work: Symbolic execution and program specialization have been used together in previous work. In [6], symbolic execution is used to achieve program specialization of Ada programs. In contrast, our approach uses program specialization to achieve compositional symbolic execution of object-oriented Java Bytecode. In [3], symbolic execution and partial evaluation are interleaved to speed up a logic-based verification framework. However that work does not address compositionality in symbolic execution, which is the focus here.

2. Background: Symbolic Execution and Symbolic PathFinder

Symbolic Execution [10, 5] is a programming analysis technique which executes programs with unspecified inputs, by using symbolic inputs instead of concrete ones. For each executed program path, a path condition is built which represents the condition on the inputs for the execution to follow that path, according to the branching conditions in the code. The satisfiability of the path condition is checked at every branching point, using off-the-shelf solvers. Thus only feasible paths are explored. A symbolic execution *tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program (symbolic) states and the arcs represent transitions between states. Test case generation and error detection are among the most studied applications of symbolic execution, with several tools available [4].

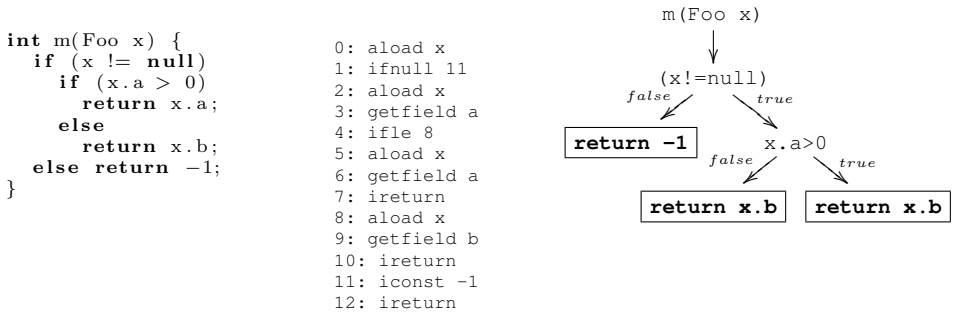

Figure 1: Java source code, bytecode and symbolic execution tree (sketch)

Symbolic PathFinder (SPF) is a symbolic execution framework built on top of the Java PathFinder (JPF) model checking toolset for Java bytecode analysis. SPF implements a bytecode interpreter that replaces the standard, concrete execution semantics of bytecodes with a non-standard symbolic execution. Non-deterministic choices in branching conditions are handled by means of JPF's *choice generators*. JPF's listeners are used to monitor and influence the symbolic execution and to collect and print its results. Symbolic execution of looping programs may result in an infinite symbolic execution tree; for this reason, SPF is run with a user-specified bound on the search depth.

SPF uses *lazy initialization* [9] to handle dynamic input data structures (e.g., lists and trees). The components of the programs inputs are initialized on an "as-needed" basis. The intuition is as follows. To symbolically execute method `m` of class `C`, SPF creates a new object `o` of class `C`, leaving all its fields uninitialized. When a reference field `f` of type `T` is accessed in `m` for the first time, SPF non-deterministically sets `f` to `null`, a new object of type `T` with uninitialized fields, or an alias to a previously initialized object of type `T`. This enables the systematic exploration of different heap configurations during symbolic execution.

Fig. 1 shows the Java source code and bytecode for a method `m`, together with the symbolic execution tree (sketched).

3. Compositional Symbolic Execution

3.1. Method Summaries

A summary for a method m is a succinct representation of all the terminating paths of its symbolic execution tree. We impose a depth limit in the tree to ensure its finiteness. For each terminating path, we store a *summary case*, i.e., a quadruple $\langle PC, HPC, C, S\rangle$, where PC is the path condition, HPC is the heap path condition, C is a specialized version of the bytecode of m, and S is the associated composition schedule, as described in more detail below.

The path condition is a conjunction of constraints over the symbolic numeric input arguments of the method or numeric fields in the heap. These constraints

Case	PC	HPC	Code	Schedule
1	$\emptyset$	$\{x = null\}$	[iconst -1, ireturn]	[ ]
2	$\{x.a > 0\}$	$\{x \neq null\}$	[aload x, getfield a, ireturn]	[ ]
3	$\{x.a \leq 0\}$	$\{x \neq null\}$	[aload x, getfield b, ireturn]	[ ]

Figure 2: Method summary

are generated and checked for satisfiability (using an off-the-shelf solver) during the symbolic execution of conditional bytecode instructions such as `ifle`, `if_icmpeq`, `if_acmpeq`, `dcmpg`, etc.

The heap path condition is a conjunction of constraints over the heap allocated objects in the input data structure. These constraints are generated by lazy initialization during the symbolic execution of instructions `aload`, `getfield` and `getstatic`. The constraints can have the following form:

- $Ref = null$. Reference *Ref* points to *null*;
- $Ref \neq null$. Reference *Ref* is a new object, different from *null* and not aliased with any other object in the heap, with all its fields made symbolic;
- $Ref_1 = Ref_2$. References Ref_1 and Ref_2 are aliased, i.e., point to the same object in the heap.

In an extended version of this paper we will show that these constraints are sufficient to express all the possible aliasing scenarios in the input data structures. The specialized code corresponds to the code executed along a particular path in the program. All forms of non-determinism are resolved and the specialized code is guaranteed to contain no branching conditions (see Section 3.2). Thus, when a summary is (re-)used during compositional symbolic execution, no expensive constraint solving needs to be performed *during* the execution of the specialized code.

The composition schedule allows for incremental, deterministic composition of method summaries by determining, for each invoke instruction in the specialized code, which case from the invoked method's summary must be composed.

Fig. 2 presents the summary for method `m` in Fig. 1. The first element corresponds to the case in which `x` is lazily initialized to `null` and the path condition remains empty. The second and third cases correspond to `x` being lazily initialized to a new object, for different PCs. The case 2 represents the path corresponding to the *then* branch of the inner if statement and case 3 represents the *else* branch.

Note that since the execution tree may be infinite, a termination criterion is needed to ensure finiteness of the process. A summary is a finite representation of the symbolic execution of a program with respect to the given termination criterion. In other words, a summary is a complete specification of the program for the given criterion, but still a partial specification in general.

3.2. Program Specialization during Symbolic Execution

Algorithm 1 formalizes how we use partial evaluation to generate method summaries. Procedure Specialization is applied to each instruction in the method,

Algorithm 1 Specialization during Symbolic Execution

```
Input:   insn:Instruction, currentState ≡ ⟨ pc, hpc, code, sched ⟩
 1: procedure SPECIALIZATION
 2:     switch TYPE(insn) do
 3:         case ConditionalInstruction
 4:             code ← SLICECODE(code,insn)
 5:         case InvokeInstruction
 6:             COMPOSESUMMARY(getInvokedMethod(insn),duringSpecialization)
 7:             code ← APPEND(code,insn)
 8:         case ReturnInstruction
 9:             code ← APPEND(code,insn)
10:             STORESUMMARYCASE(pc,hpc,code,sched)
11:         case GotoInstruction
12:             IGNORE
13:         default
14:             code ← APPEND(code,insn)
```

during symbolic execution. Conditional instructions (`ifle`, `ifnull`, ...) are left out of the specialized code, together with the instruction(s) that pushed the operand(s) for the conditionals (procedure SLICECODE). For invoke instructions, the summary for the invoked method is used (see next section), and the instruction is appended to specialized code. Return instructions are appended to the specialized code and lead to the addition of a new summary case. The remaining instructions, except the `goto` instruction which is ignored, are just appended to the specialized code.

For example, the case 3 in Fig. 2 is obtained by performing partial evaluation along the symbolic path in method `m` (Fig. 1) with PC $x.a > 0$ and HPC $x \neq null$. Instructions `aload`, `getfield` and `ireturn` are simply appended to the specialized code (line 14 in procedure SPECIALIZATION), while instructions `aload x` and `ifnull 11` are removed (line 4). Later on, when the conditional `ifle 8` is executed, a special case in procedure SLICECODE is activated, which removes not only the previous pushing instruction `getfield`, but also the complete reference chain. Section 4 reports on a case study that unveils the benefits of this mechanism to produce shorter and more efficient specialized code.

3.3. Composing Summaries

When deciding on the order to analyze the methods in a given system, two main strategies can be followed. A context-sensitive or top-down approach may be adequate if we want to compute only the strictly necessary information (method summaries). However, this approach does not guarantee that reusability of summaries is always possible. On the other hand, a context-insensitive or bottom-up approach ensures that the computed summaries can always be reused, at the price of computing summaries larger than necessary in some cases. We follow the latter strategy in our framework. The methods are therefore processed in an order corresponding to a bottom-up traversal of the program's call graph, starting with the ones that invoke no other methods and incrementally processing methods whose sub-methods already have summaries until the whole program is analyzed. We assume no recursion, which is left for future work.

ComposeSummary: Let us assume that a method `comp` invokes method `m`. By construction, the existence of a method summary for `m` is assumed. Procedure composeSummary in Algorithm 2 executes every time a method invocation (e.g., `invokevirtual`) is reached. The procedure distinguishes two cases, according to parameter *mode*. When *mode* is set to *duringSpecialization* (from Algorithm 1), it means we are in the middle of creating a summary for a method (e.g., `comp`) and an invocation to another method for which a summary exists is reached (e.g., `m`). Here, for each case in the method summary for `m`, its composition schedule is established and the composition operation (procedure composeCase) is applied. Alternatively, when *mode* is set to a value different than *duringSpecialization*, it means that we are in the middle of executing specialized code and we reached a method invocation. Therefore, a deterministic composition schedule that specifies which summary case to use for the invoked method has already been set. E.g., during execution of a summary case of method `m`, an invocation to method `q` is reached; relying on the composition schedule, we can uniquely select the summary case of `q` that must be composed. Thus, there is no branching during the execution of a summary case for `m`.

ComposeCase: The composeCase procedure first analyzes the compatibility of the heap path conditions (procedure checkAndSet). This analysis implies: a) checking that the heap constraints from the summary case hold in the current concrete heap; and b) firing "eager" lazy initialization for all the heap constraints that operate on symbolic heap elements. If checkAndSet succeeds, the algorithm conjoins the path condition from the summary case into the current path condition. If the new path condition passes the satisfiability check, the original code of the invoked method is replaced with the specialized code from the summary case and symbolic execution proceeds; note that in this case *mode* is different from *duringSpecialization*, so on an invoke, the second case of composeSummary is used.

Consider case study 2 in Fig. 3. One of the 22 summary cases obtained for method `q` (the 6th) is as follows: $\langle\{x.f \geq 0,\ x.f \neq 0\},\ \{x \neq null,\ x.next = x\},$ [`aload x`, `getfield next`, `getfield next`, `getfield next`, `getfield f`, `invoke abs`, `ireturn`], $[0]\rangle$. The composition schedule [0] indicates that when processing the invocation to `abs`, the first case (index 0) of `abs`'s summary must be deterministically chosen for composition. Now, let us consider building the summary for method `m` and let us look at the symbolic execution path of `m` in which arguments `x` and `y` are lazily initialized to `null` and method `q` is invoked with argument `z` lazily initialized to a new object with symbolic fields. The composition of the 6th summary case for `q` generates the following summary case for `m`: $\langle\{z.f \geq 0, z.f \neq 0\}, \{x = null, y = null, z \neq null, z.next = z\}, [\ldots], [6, 0]\rangle$. Observe that fields `z.f` and `z.next` are constrained according to the (heap)path condition from the composed summary case. Moreover, notice that the composition schedule stored in the new summary case for `m` specifies which summary cases to be selected in further calls. Namely, during execution of this particular summary case for `m`, when the call to method `q` is reached, its summary case with index 6 will be deterministically selected; and in turn, when the call from `q` to `abs` is reached, its summary case with index 0 will be chosen.

Algorithm 2 Composition Operations

```
1: procedure composeSummary(m,mode)
2:     if mode = duringSpecialization then
3:         S ← getSummary(m)
4:         for all case ∈ S do
5:             setCompositionSchedule(case.getCompSched())
6:             composeCase(case)
7:     else
8:         S ← getSummary(m)
9:         caseIndex ← compositionSchedule.getNext()
10:        case ← getSummaryCase(S,caseIndex)
11:        composeCase(case)
12: procedure composeCase(case)
13:     heapPC ← case.getHeapPC()
14:     projectActualParameters(heapPC)
15:     if checkAndSet(currentHeapPC,heapPC) then
16:         pc ← case.getPC()
17:         projectActualParameters(pc)
18:         currentPC ← currentPC ∪ pc
19:         if satisfy(currentPC) then
20:             ReplaceCode(invokedMethod,case.getCode())
21:             ContinueSymbolicExecution                ▷ mode ≠ duringSpecialization
22:         else
23:             Backtrack
24:     else
25:         Backtrack
```

4. Experience

Our compositional approach has been implemented as an extension of SPF through two listeners that perform specialization and composition by monitoring the symbolic execution and reacting upon execution of method calls and returns. We have applied our implementation to two small case studies (Fig. 3). The first one stresses the use of linear integer constraints and was used in previous work [1]. It contains 4 methods: `abs`, `gcd`, `simplify` and `simp`, for which 2, 13, 14 and 2744 summary cases are generated, respectively. The second case study illustrates the potential impact of our approach on object-oriented programs; the code was created to illustrate the code slicing in the presence of heap updates. Here, a total of 22 and 9938 summary cases are generated for methods `q` and `m`. Table 1 compares the results obtained with standard (SPF) and compositional symbolic execution (CompSPF). Although in early stage of development, our compositional framework outperforms non-compositional SPF, both in computation time and number of explored states. The gains in number of instructions executed is quite different between the two examples; the specialized programs computed in Section 3.2 tend to be much shorter in the presence of heap constraints. This is due to the removal of many instructions by cutting off complete reference chains when specializing conditionals over reference fields. We further notice that our approach is more expensive in terms of memory cost; this is not surprising given the large number of cases that need to be stored. We are working on optimizing our implementation to alleviate this cost, e.g., by storing only bytecode references in the summaries and by simplifying the numeric constraints.

```
int abs(int x){
  if (x >= 0) return x;
  else return -x;
}
int gcd(int x, int y) {
  if (x == 0) return abs(y);
  while ((y != 0)  && (i<2)) {
    if (x > y) x = x-y;
    else y = y-x;
    if (i==2) return -1;
    i++;
  }
  return abs(x);
}
class R {
  int num, den;
  void simplify(int a, int b){
    int gcd = gcd(a,b);
    if (gcd != 0) {
      num = num/gcd; den = den/gcd;
    } }
  R[] simp(R[] rs){
    R[] oldRs = new R[rs.length];
    arraycopy(rs,oldRs,length);
    for (int i=0;i < length;i++)
      rs[i].simplify(rs[i].num,
                     rs[i].den);
    return oldRs;
  } }
```

```
class Foo {
  int f; Foo next;
}
int q(Foo x, Foo y){
  if (x != null) {
    if ((x.next != null) &&
       (x.next.next != null) &&
       (x.next.next.f != 0))
      return x.next.next.f;
    else
      return 1;
  } else if ((y != null) &&
             (y.next != null) &&
             (y.next.f != 0))
          return abs(y.next.f);
        else
          return 2;
}
void m(Foo x,Foo y, Foo z){
  Foo[] arr = new Foo[]{x,y,z};
  for (int i=0;i<arr.length;i++){
    if (arr[i] != null)
      arr[i].f = q(arr[i],y);
    else
      arr[i] = new Foo(0,0);
  }
}
```

Figure 3: Case Study 1 (left) and Case Study 2 (right)

	Case Study 1		Case Study 2	
	SPF	CompSPF	SPF	CompSPF
Time	00:02:50	00:01:02	00:00:51	00:00:13
States	24899	13928	86175	27762
Instructions	145908	139992	1215959	223786
Max. Memory	106MB	170MB	242MB	364MB

Table 1: Preliminary experimental results

5. Conclusions and Future Work

We have described a compositional approach to symbolic execution that is based on program specialization. We have reported on its implementation in SPF for the analysis of Java bytecode. Our evaluation is very preliminary. More evaluation is planned for an extended version of this paper, where we also plan to provide a more detailed treatment of loops and to prove the correctness of our approach. Furthermore we plan to make our implementation available via SPF's open-source repository. In future work, we plan to extend our approach to multithreaded programs, where the path explosion problem is even more acute. The works on *schedule specialization* [14] and *program sequentialization* [13] should be relevant there.

Acknowledgments. The work of José Miguel Rojas was funded in part by the Spanish projects TIN2008-05624 *DOVES* and TIN2012-38137 *VIVAC*.

References

[1] E. Albert, M. Gómez-Zamalloa, J. M. Rojas, and G. Puebla. Compositional CLP-based test data generation for imperative languages. In *Proc. of LOPSTR'10*, volume 6564 of *LNCS*, pages 99–116. Springer-Verlag, 2011.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. of TACAS'08*, pages 367–381. Springer-Verlag, 2008.

[3] R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In *Proc. of FMCO'09*, pages 125–146. Springer-Verlag, 2010.

[4] C. Cadar, P. Godefroid, S. Khurshid, C. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proc. of ICSE'11*, pages 1066–1071. ACM, 2011.

[5] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. on Software Engineering*, 2(3):215–222, 1976.

[6] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Trans. on Software Engineering*, 17(9):884–899, 1991.

[7] P. Godefroid. Compositional dynamic test generation. In *Proc. of POPL'07*, pages 47–54. ACM, 2007.

[8] N. D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[9] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of TACAS'03*, pages 553–568. Springer-Verlag, 2003.

[10] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] C. Păsăreanu, P. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. of ISSTA'08*, pages 15–26. ACM, 2008.

[12] C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.

[13] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. *SIGPLAN Not.*, 39(6):14–24, 2004.

[14] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proc. of PLDI'12*, pages 205–216. ACM, 2012.