Theoretical Aspects of Compositional Symbolic Execution

Dries Vanoverberghe* and Frank Piessens

Dries.Vanoverberghe, Frank.Piessens@cs.kuleuven.be

Abstract. Given a program and an assertion in that program, determining if the assertion can fail is one of the key applications of program analysis. Symbolic execution is a well-known technique for finding such assertion violations that can enjoy the following two interesting properties. First, symbolic execution can be *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Second, it can be *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure.

Recently, *compositional* symbolic execution has been proposed. It improves scalability by analyzing each execution path of each method only once. However, proving precision and progress is more challenging for these compositional algorithms. This paper investigates under what conditions a compositional algorithm is precise and progressing (and hence a semi-decision procedure).

Keywords: Compositional, symbolic execution, precision, progress.

1 Introduction

Given a program and an assertion in that program, determining whether the assertion can fail is one of the key applications of program analysis. There are two complementary approaches.

One can try to determine whether the assertion is *valid*, i.e. is satisfied in all executions of the program. This can be done using techniques such as type systems, abstract interpretation, or program verification. Such techniques are typically expected to be *sound*: if they report an assertion as valid, there will indeed be no execution that violates the assertion. However, these techniques suffer from false positives: they may fail to establish the validity of an assertion even if there is no execution that violates the assertion.

Alternatively one can look for counterexamples by trying to determine inputs to the program that will make the assertion fail. One important technique for this approach is symbolic execution [1], a well-known analysis technique to explore the execution traces of a program. The program is executed symbolically using logical symbols for program inputs, and at each conditional the reachability of both branches is checked

* Dries Vanoverberghe is a research assistant of the Fund for Scientific Research - Flanders (FWO).

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 247–261, 2011.

using an SMT solver. When reaching the assertion, the analysis determines if it can find values for the symbolic inputs that falsify the assertion. Such a technique can not prove the validity of an assertion, but it has the advantage of avoiding false positives (a property that we will call *precision*). Obviously, sound and precise approaches are complementary. This paper focuses on precise algorithms, and more specifically on precise symbolic execution.

Thanks to many improvements to SMT solvers, symbolic execution has become an important technique, both in research prototypes [2,3,4,5,6,7,8,9,10] as well as in industrial strength tools [3,4]. Recently, *compositional* symbolic execution [11,12] attempts to further improve the scalability of symbolic execution. With compositional symbolic execution, each execution path of a method is only analyzed once. The results of this analysis are stored in a so-called *summary* of the method, and are reused by all callers of the method.

Traditional whole-program (non-compositional) symbolic execution has two interesting properties that are not necessarily maintained in the compositional case. First, as discussed above, symbolic execution can be *precise*: if it reports that an assertion can fail, then there is an execution of the program that will make the assertion fail. Proving precision for whole-program symbolic execution is relatively easy: one has to prove that symbolic execution correctly abstracts concrete executions, and that the SMT solver is sound and complete (which it can be for the class of constraints it needs to solve). Second, symbolic execution can *make progress* or be *progressing*: if there is an execution that makes the assertion fail, it will eventually be found. Therefore, there are no classes of programs where the analysis fails fundamentally. Again, making a symbolic execution algorithm progressing is relatively straightforward, for instance by making the algorithm explore the tree of possible paths through the program in a breadth-first manner. Since this tree is finitely-branching, a breadth-first exploration ensures that any node of the tree will eventually be visited. A symbolic execution algorithm that is both precise and progressing is a semi-decision procedure for the existence of counterexamples.[1]

Although compositional symbolic execution is inspired by standard symbolic execution, the proofs of these important properties become much more challenging. In fact, some of the algorithms proposed recently are not necessarily semi-decision procedures. This paper develops proof techniques for showing precision and progress of compositional symbolic execution algorithms.

More specifically, this paper makes the following contributions:

- We formally model the existing compositional symbolic execution, based on a small but powerful programming language.
- We show that any compositional symbolic execution algorithm based on this formal model is *precise*.
- We give sufficient conditions for an algorithm to be *progressing*, and therefore be a semi-decision procedure.

[1] Note that precision is a soundness property, and progress is a completeness property, but we avoid the terms soundness and completeness on purpose to avoid confusion with soundness and completeness of verification algorithms or theorem provers.

For the purpose of investigating precision and progress, the assertion in the program is not relevant. What matters is whether the symbolic execution algorithm correctly enumerates all the reachable program states. Hence, for the rest of this paper, we will consider symbolic execution algorithms to be algorithms that enumerate reachable program states. Such an algorithm is precise if any program state that it enumerates is also reachable by the program. It is progressing if any program state reachable by the program is eventually enumerated.

The rest of this paper is structured as follows. First, in Section 2 we show by means of examples that precision and progress are hard to achieve for compositional algorithms. Then we introduce a small but powerful programming language in Section 3. Section 4 presents compositional symbolic execution and creates a formal model of it based on transition systems. Next, we show that this algorithm is precise and progressing (Section 5). Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Motivation

Traditional symbolic execution [1] explores paths through the program by case splitting whenever the execution reaches a branch. Since loops are also just branches that are encountered multiple times, this implies that loops are lazilly unrolled, potentially an infinite amount of times[2]. When a method call is reached, the target method is symbolically executed using the given arguments. Therefore, if the program calls a given method several times, the execution paths in that method will be re-analyzed for each call. The key idea of compositional algorithms is to avoid this repeated analysis. Instead, execution paths are explored for each method independently. The results of this exploration are stored in a *method summary*. Method calls are no longer inlined: a method call is analyzed in one single step and the result is computed based on the summary of the target method. Compositional symbolic execution has been shown [11,12,13] to improve performance, but maintaining precision and progress is challenging.

2.1 Precision

Compositional symbolic execution creates two potential causes of imprecision. First, when there is insufficient information about the calling context of a method, then one might conclude that unreachable program locations are reachable. For example, the highlighted statement in the method $P2$ in Figure 1 is unreachable in the current program because the method $P1$ only calls $P2$ with argument $x \mathrel{!}= 0$. However, if one would analyze $P2$ independently of $P1$, the analysis might conclude that the highlighted statement is reachable. In other words, since reachability is a whole-program property, we need to maintain some whole-program state even in a compositional analysis. The example algorithm we discuss later will do so by maintaining an invocation graph.

[2] Developer provided or automatically synthesized invariants can be used to create sound analyzers for particular classes of programs. This paper considers the general case where such invariants are not present or cannot be inferred.

Second, when a method returns and the analysis loses information about the relation between the arguments of the method and the return value, then the analysis might incorrectly conclude that a program location is reachable. For example, the highlighted statement in the method $P1$ in Figure 1 is unreachable. When the analysis over-approximates the result of $P2$ by the relation $result == 0 \vee result == 1 \vee result == -1$, then the highlighted location is reachable. To maintain precision, method summaries should not introduce such approximations.

```
int P1(int x) {
  if(x != 0){
    int r2 = P2(x);
    if(x > 0 && r2 != 1) return -1;
  }
  return 0;
}
```

```
int P2(int u) {
  if(u == 0) return 0;
  else if(u > 0) return 1;
  else return -1;
}
```

Fig. 1. Example program for precision

2.2 Progress

Non-compositional symbolic execution builds one global execution tree where leaf nodes represent either final program states, unreachable program states, or program states that require further analysis. Given a fair strategy to select such leaf nodes for further analysis, it is easy to show that the depth of the highest unexplored node keeps increasing and hence that any finite execution path will eventually be completely analyzed. This implies progress for non-compositional symbolic execution.

For compositional symbolic execution, the situation is more complex due to two reasons. First, as we discussed above, in order for method summaries to be precise, they must depend on the calling context. Hence, the discovery of a new call site for a method may increase the number of reachable points in the method and unreachable leaf nodes need to be reanalyzed taking into account the new calling context.

Secondly, when analyzing a method call, a compositional analysis relies on the summary of the target method for computing the return value. However, method summaries change over time when the analysis discovers new returns. As a consequence, nodes that were deemed unreachable based on the summary of the method must be reanalyzed when that method summary is updated.

The progress argument for non-compositional symbolic execution relies essentially on the fact that unreachable leaf nodes remain unreachable for the rest of the analysis. With compositional symbolic execution, this premise is no longer satisfied. Furthermore, it is impossible to guarantee that any finite execution path within the execution tree of a single method will eventually be completely analyzed. The program in Figure 2 provides an example of this phenomenon.

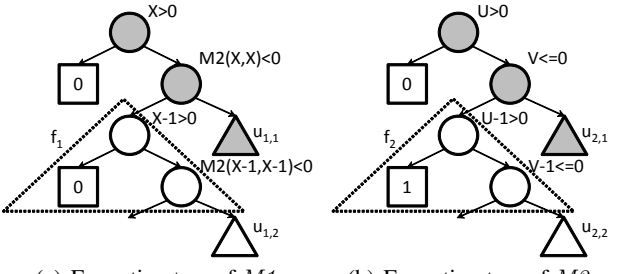
First, we explain the program: The two highlighted statements are both unreachable, and therefore the method $M1$ returns 0 for any input. To understand this, two invariants are important: First, the method $M1$ only calls the method $M2$ with parameters $u = v$ with $u > 0$. Second, if the parameters u and v of $M2$ are greater than zero, then $M2$ returns the minimum of u and v.

```
int M1(int x) {
  while (x > 0) {
    int y = M2(x, x);
    if(y < 0) return −1;
    x−−;
  }
  return 0;
}
```

```
int M2(int u, int v) {
  int w = 0;
  while (u > 0) {
    if(v <= 0) return − w;
    u−−; v−−; w++;
  }
  return w;
}
```

Fig. 2. Example program for progress

Figures 3(a) and 3 show the execution trees of $M1$ and $M2$. Each circle represents a case split in the program, and the corresponding condition is written in the upper-right corner. From a circle, the arc to the left (right) means that the condition is false (true). Squares are final nodes, and imply that the method returns with the return value written inside the square. Triangle denotes unreachable nodes.



(a) Execution tree of $M1$ (b) Execution tree of $M2$

Fig. 3. Execution trees

Let $u_{i,j}$ be the analysis step that checks the j-th unreachable node of Mi, and f_i the sequence of analysis steps that explores the reachable part of the execution tree of Mi. The sequence f_1 causes a new invocation from $M1$ to $M2$ and therefore resets the unreachable nodes in $M2$. The sequence f_2 causes a new return and therefore resets the unreachable nodes in $M1$. Let $u_{i,2..}$ be the sequence $u_{i,2}, \ldots, u_{i,n}$ where $u_{i,n}$ checks the deepest unreachable node of Mi. Suppose we analyze according to the fair schedule $f_1, f_2, [u_{2,1}, f_1, u_{2,2..}, u_{1,1}, f_2, u_{1,2..}]^*$. Then the highlighted nodes in the execution trees, that are only at depth 3 in the tree will be of status needs-further-analysis an infinite number of times. Hence, the depth of the analysis never stays larger than 3, and the given schedule is a counter example for the traditional proof of progress.

This shows that progress for compositional symbolic execution can not be proved by mimicking the proof for the non-compositional case on a per-method basis. In Section 5, we will propose an alternative technique to prove progress for compositional symbolic execution.

3 Programming Language

In this section, we introduce a small intermediate language that is particularly well-suited for presenting compositional symbolic execution. It only retains the structure

of the program that is essential: the structure of the control flow graph per procedure, and the calls and returns between procedures. The language focuses on sequential programs. Besides this restriction, all relevant more complicated language features can be translated to this core (e.g. parameters, return values or loops, ...). For brevity, we also assume that the program does not contain (mutually) recursive methods. Although the algorithm in 4 depends on this simplification, our prototype implementation supports recursion by inferring ranks, but the details are beyond the scope of this paper.

A program p is a tuple $\langle M_p, G_p, m_p^0 \rangle$ where M_p is a set of methods, G_p is a set of global variables and $m_p^0 \in M_p$ is a distinguished entry method. Each method definition m for the program p is a tuple $\langle L_m, N_m, \lambda_m, n_m^0 \rangle$ where L_m is a finite set of local variables disjoint from the global variables G_p, N_m is a finite set of program locations, $n_m^0 \in N_m$ is a distinguished entry node and $\lambda_m : N_m \rightarrow Commands_{m,p}$ maps each node to a command. The sets of local variables and nodes of different methods are disjoint.

A command c for the method m of the program p is either:

- An assignment $\mathbf{assign}\ x, e, n$ where $x \in L_m \cup G_p$, e is a side-effect free expression over $L_m \cup G_p$ and constants, and $n \in N_m$ is a program location. This command updates the value of the variable x, and continues in location n.
- A conditional $\mathbf{if}\ e, n_t, n_f$ where e is a side-effect free expression over $L_m \cup G_p$, and $n_t, n_f \in N_m$ are program locations. If the expression e evaluates to true (false) the execution continues in location n_t (n_f).
- A call $\mathbf{call}\ m_t, n$ where $m_t \in M_p$ is the target method and $n \in N_m$ is a program location. This command invokes the method m_t and continues in location n.
- A return $\mathbf{ret}$ returns from the current method.

For each variable v, $\mathcal{D}(v)$ represents the value domain of the variable. A valuation σ_V is a partial function that maps each variables $v \in V$ to a value $val \in \mathcal{D}(v)$. Each domain has a default element $\mathcal{D}_0(v)$, and the default valuation σ_V^d for a set of variables V maps each variable $v \in V$ to $\mathcal{D}_0(v)$.

An execution state $s \in S_p$ for the program p is tuple $\langle \sigma_G, \overline{f} \rangle$ where

- σ_G is the current valuation for G_p
- $\overline{f} \in F_p^*$ is a sequence of frames for p (sequences are either empty (nil) or a concatenation $h; \overline{t'}$ of a head h and a tail $\overline{t'}$)

A frame $f \in F_p$ for the program p is a tuple $\langle m, n_m, \sigma_{L_m} \rangle$ where

- $m \in M_p$ is the current method.
- $n \in N_m$ is the current program location.
- σ_{L_m} is the current valuation for L_m

The operational semantics $\rightarrow \subseteq S \times S$, gives an interpretation to the commands (More details can be found in a technical report [14]), and $\rightarrow^*$ is its reflexive transitive closure. The execution of a program p starts in the initial state $s_p^0 = \langle \sigma_{G_p}^0, f_{m_p^0}^0 \rangle$ with $f_{m_p^0}^0$ the initial frame for m_p^0, and $\sigma_{G_p}^0$ the input valuation for the global variables G_p. The initial frame for a method m is $f_m^0 = \langle m, n_m^0, \sigma_{L_m}^d \rangle$.

A state s is reachable from a state s' if and only if $s \rightarrow^* s'$. A state s is reachable in a program pr (denoted as $\models reach(pr, s)$) if and only if s is reachable from the initial state s_{pr}^0 of the program.

4 Compositional Symbolic Execution

In this section, we model the essence of existing compositional symbolic execution algorithms in order to formally study their precision and progress properties. Symbolic execution [1] is a technique to explore the execution paths of a program under all possible inputs. Instead of using a concrete input, the execution of the program is started with symbols representing arbitrary values. As a result, the values in the symbolic execution state are symbolic expressions that depend on the input symbols. Symbolic interpretation lifts the interpretations of commands to symbolic states.

For each execution path, symbolic execution constructs a *path condition*, a constraint in function of the input symbols that characterizes when an input follows that path. At a branch with condition C, the result of concrete execution is statically unknown: Either C is true and execution continues with the true-branch, or C is false and the false-branch is taken. Therefore, symbolic execution splits the set of inputs in two new sets: one where C is added to the path condition and one where the negation of C is added to the path condition. To check reachability, i.e. whether there is an execution that follows a path, a constraint solver checks satisfiability of the path condition. As a result, symbolic execution explores a prefix of the (potentially infinite) execution tree of the program. The resulting prefix is a partition of the global input space of the program.

The difference between compositional [11,12] and traditional symbolic execution is in the treatment of method calls and returns. In traditional symbolic execution, a call adds a new symbolic frame for the target method and continues execution until a return command pops this frame. Therefore, when a method is called twice, a path through the method is computed twice, even if it is guaranteed to follow the same path. Compositional symbolic execution explores the execution trees of each method in isolation. This results in a partition for each method (also called the summary). When a call is encountered, compositional symbolic execution uses the summary of the target method to compute the effect on the symbolic state.

To show that compositional symbolic execution is a semi-decision procedure, it is convenient to model the algorithm as a transition system $a \Rightarrow a'$ (and $\Rightarrow^*$ its reflexive transitive closure) which starts in an initial analysis state a^0. Non-terminating runs of the algorithm can be truncated after any number of transitions. In addition, the predicate $\vdash^a reach(p, s)$ denotes that the analysis concludes the reachability of the state s in the program p in an analysis state a.

Such a transition system is precise if and only if the conclusion in any reachable analysis state is sound[3]:

Definition 1 (Precision). *For each program pr, concrete state s, and analysis state a such that $a^0_{pr} \Rightarrow^* a$, $\vdash^a reach(pr, s)$ implies $\models reach(pr, s)$.*

Obviously, compositional symbolic execution is not complete[4] in any reachable analysis state and due to undecidability this is even impossible. However, the analysis incrementally discovers more and more reachable states. This incremental nature is captured by monotonicity:

[3] Sound as a bugfinder, i.e. any state which is concluded reachable is truly reachable.
[4] Complete as a bugfinder.

Definition 2 (Monotonicity). *For each program pr, concrete state s, and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^{a} reach(pr, s)$ then $\vdash^{a'} reach(pr, s)$.*

For a monotonous analysis, progress is the next best thing with respect to completeness: for any reachable concrete state, eventually there is an analysis state that concludes reachability for that concrete state:

Definition 3 (Progress). *For each program pr and each concrete state s, if $\models reach(pr, s)$ then for all analysis states a' such that $a^{0}_{pr} \Rightarrow^{*} a'$ there exists an analysis state a such that $a' \Rightarrow^{*} a$ and $\vdash^{a} reach(pr, s)$. In other words, for each reachable concrete state s, there always eventually is an analysis state that concludes s is reachable.*

When an analysis is precise, monotonous and progressing, it is a semi-decision procedure.

4.1 Overview

The analysis state maintains a summary per method, which is a set of leaf nodes of the current prefix of the execution tree of the method. A leaf node $\langle stat, \nu, pc \rangle$ contains:

- A status $stat$, which is either unknown, finished or unreachable,
- A symbolic execution state ν,
- A path condition pc.

The path condition defines the inputs (i.e. the values of the global variables) that will drive the execution of the method along this path. The symbolic execution state represents the state of execution after executing the path. The status indicates whether (a) the path is a complete path through the method, i.e. the method returns after this path (finished status) (b) the path is unreachable (unreachable status) (c) further exploration of continuations of this path are needed (unknown status). Symbolic execution states are defined like concrete execution states, except that all valuations are symbol valuations i.e. any variable has a symbolic expression instead of a concrete value.

The summaries only maintain per-method information. As we have shown in Section 2, it is necessary to maintain some whole program information in order to be precise. In particular, it is important to precisely track reachable method invocations and returns.

Initially, only the main method is reachable. As the analysis progresses, any call that is discovered is stored in an invocation graph. This graph is represented as a set of invocations, where each invocation is a tuple $\langle m_s, m_t, \varsigma_G, pc \rangle$. The methods m_s and m_t are the source and target methods, and ς_G and pc are the symbolic values of the global variables and the path condition at the moment of the invocation. Reachability checking will use the information in the call graph to decide whole-program reachability.

To support discovery of new returns efficiently, the return values of method calls can be modeled using logic function symbols[12]. The symbolic execution of a call is defined in terms of these function symbols. The interpretations of the function symbols are constructed using the current summary. As the analysis progresses, they become precise for more and more inputs. We discuss this in more detail in Section 4.2.

In addition, the analysis tracks all reachable program states it has enumerated. For this purpose, the analysis state contains a set of leaf nodes that succeeded the reachability check for each method. Based on this information, reachability conclusion is defined. If a leaf node $\langle stat, \nu, pc \rangle$ is in the reachable set of the method m in a given analysis state a, then any concretization of its symbolic state ν with global variables satisfying pc is concluded reachable in m. A state s is concluded reachable in an analysis state a (denoted $\vdash^a reach(pr, s)$) if and only if either

- s is concluded reachable in the entry method m^0_{pr} in a or
- there is a state s' such that $\vdash^a reach(pr, s')$ and s' calls m and s is reachable in m.

Usually, one is only interested whether a point in a program is reachable (e.g. a location n in a method m). Therefore, implementations often store reachable program points instead of leaf nodes or avoid the reachability set completely by reporting an error when reaching a distinguished error-location. However, reachability conclusion of arbitrary states is essential for inductive invariants that enable the precision and progress proofs.

Finally, an analysis state can be defined as a tuple $\langle sum, invs, rs \rangle$ where

- sum is a function that maps each method m to a set of leaf nodes (its summary).
- $invs$ is a set of invocations.
- rs is a function that maps each method to a set of reachable leaf nodes.

In the initial analysis state a^0_p, the invocation graph and the sets of reachable leaf nodes are empty. Each summary starts with a symbolic execution state at the entry of the method, where the value of all global variables contains a new symbol.

The high level overview of one step of the compositional symbolic execution algorithm is shown in Figure 4. During each step, the algorithm chooses a method m and an leaf node $\epsilon \in sum_a(m)$ with unknown status. Then, the algorithm checks whether there exists an input $\sigma^0_{G_p}$ such that the execution enters the method m and the global variables satisfy the path condition pc_ϵ of ϵ ($Check(a, m, \epsilon.pc)$). If there is no such input, the status of the ϵ is changed to unreachable. Otherwise, ϵ is added to the set of reachable leaf nodes of m, and symbolic execution continues with the interpretation ($SyInt(a, m, \epsilon)$) of the symbolic state ν_ϵ of ϵ. When symbolic interpretation finishes, it returns a set of new equivalence leaf nodes, and the current leaf node ϵ is replaced by the new leaf nodes ($ReplaceLeaf$). All method calls in this algorithm are guaranteed to terminate, and therefore one step of the algorithm always terminates.

We now zoom in on some aspects of the algorithm that are of importance for precision and progress.

4.2 Symbolic Interpretation

In this section, we informally discuss the inference rules for symbolic interpretation ($SyInt$). Full details are included in a technical report [14].

As pointed out in the previous section, the analysis uses uninterpreted function symbols to support discovery of new returns as the analysis progresses. The algorithm models the effect of the method m on the global variable v as an uninterpreted function symbol $rv_{m,v}$. When a method m is called with global variables ς_{G_p}, then the function application $rv_{m,v}(\varsigma_{G_p})$ models the value for the global variable v after executing m.

```
AnalysisState Step(AnalysisState a) {
  (m, ϵ) = Choose(a);
  if(Check(a, m, ϵ.pc)) {
    a′ = AddReachable(a, m, ϵ);
    (a″, π) = SyInt(a′, m, ϵ);
    return ReplaceLeaf(a″, m, ϵ, π);
  } else {
    return MarkUnreach(a, m, ϵ);
  }
}
```

Fig. 4. High level algorithm of symbolic execution

In addition, the method summaries are *partial*: there is no information about unexplored paths through a method. To deal with this, the algorithm models the set of global variable valuations that follow a finished path as an uninterpreted predicate rc_m.

During reachability checking, the algorithm computes the interpretation for the uninterpreted symbols using the method summaries (Figure 5) and replaces them using substitution.

$$
\begin{aligned}
&interps(sum) = \textstyle\bigcup_{m \in M_p} interp(m, \{\epsilon | \epsilon \in sum(m), stat_\epsilon = \mathit{fin}\}) \\
&interp(m, \pi) = rc_m \mapsto interp_{rc}(m, \pi) \cup (\textstyle\bigcup_{v \in G_p} rv_{m,v} \mapsto interp_{rv}(m, v, \pi)) \\
&interp_{rc}(m, \pi) = \textstyle\bigvee_{\langle \mathit{fin}, \nu, pc \rangle \in \pi} pc \\
&interp_{rv}(m, v, \emptyset) = \mathcal{D}_0(v) \\
&interp_{rv}(m, v, \langle \mathit{fin}, \nu, pc \rangle \cup \pi) = ite\ pc\ \varsigma_{G_\nu(v)}\ interp_{rv}(m, v, \pi)
\end{aligned}
$$

Fig. 5. Interpretation of uninterpreted function symbols

For precision, it is essential that the interpretation of the uninterpreted symbols is precise: Whenever the return condition $rc_m(\sigma_{G_p})$ is true, the execution of the method m starting with global variables σ_{G_p} eventually reaches a return command, and each global variable v must equal $rv_{m,v}(\sigma_{G_p})$.

The treatment of assignment and branches is similar to the treatment in non-compositional symbolic execution: For an assignment, symbolic interpretation performs the same operation but on symbolic expressions instead of concrete values. For branches, symbolic interpretation creates a new leaf node for each branch and conjoins the branch condition or its negation to the path condition.

The rule call creates a new leaf node where the return condition is added to the path condition, and the return values are used to update the global variables. As mentioned in Section 2, some leaf nodes can become reachable by performing a call, and progress requires that all such leaf nodes are reconsidered. The algorithm conservatively reconsiders all unreachable leaf nodes of methods that are transitively reachable in the invocation graph by marking them as unknown (using the function $rec(a, m)$, defined more precisely in Appendix). In practice, more intelligent re-evaluation strategies can take the context into account in order to minimize the number of affected nodes, but this is beyond the scope of the formal model.

The return rule marks the unknown leaf node as finished, and thereby the interpretations of the current method change. In addition, the return rule marks all unreachable leaf nodes that depend on the return condition as unknown again (using the function $rer(a, m)$, also defined in Appendix). This is again essential to maintain progress.

For precision, the symbolic interpretation algorithm must maintain precision of the leaf nodes, i.e. if an input is a member of a leaf node, then the execution starting with that input eventually reaches the concretization of the symbolic state (the symbolic state after replacing the input symbols with the concrete input). In addition, all invocations $\langle m_s, m_t, \varsigma_G, pc\rangle$ in the invocation graph must be precise: If an input satisfies the condition pc, then the execution of m_s starting with that input reaches a call to the method m_t and the global variables are the concretization of ς_G.

For progress, it is essential that symbolic interpretation maintains *totality* of the summaries. A reachable concrete state s is on the frontier if all predecessors in the execution to s are concluded reachable, but s is not concluded reachable. The summaries are total if any concrete state s on the frontier is a concretization of some unknown leaf node ϵ in the summary of some method m. Informally, this is a kind of completeness guarantee for symbolic interpretation. For any concrete state on the frontier, the analysis can make the "right" choice. Totality implies that leaf nodes may not be marked unreachable if $Check$ succeeds in the current analysis state. For this reason, the call and return rules need to reconsider some unreachable leaf nodes.

4.3 Reachability Checking

Finally, to check reachability ($Check(a, m, pc)$), the algorithm globalizes the path condition pc based on the invocation graph inv_a, substitutes the symbols $\varsigma^0_{G_p}$ by their interpretation $interps(sum_a)$, and uses an SMT-solver to check the satisfiability of the resulting constraints. The globalization $glob_p(a, m, pc)$ globalizes the constraint pc in the context of m using the invocation graph $invs_a$ and is defined inductively as follows:

- If $m = m^0_p$ then $glob_p(a, m, pc) = pc$
- If $m \neq m^0_p$ then $glob_p(a, m, pc) =$
  $\bigvee_{\langle m_s, m, \varsigma_G, pc'\rangle \in invs_a} glob_p(a, m_s, pc' \wedge pc[\bigcup_{v \in G_p} \varsigma^0_{G_p}(v) \mapsto \varsigma_{G_p}(v)])$

In the absence of recursion, the invocation graph is cycle free, and the inductive definition is well-founded.

For precision, it is important that $Check(a, m, pc)$ only returns true when there is a reachable state s where the execution enters m and the global variables satisfy pc (*precision of Check*). This follows from precision of the leaf nodes, the precision of the interpretations and the soundness of the SMT-solver as a satisfiability checker.

The contrary is not the case: If there is an execution that enters m where the global variables satisfy pc, $Check(a, m, pc)$ need not return true because this execution might follow an unexplored path through some method. For progress, it is only necessary that $Check(a, m, pc)$ holds if the execution that enters m where the global variables satisfy pc only uses concrete states that are concluded reachable (*Restricted completeness*). This requires completeness of the SMT-solver as a satisfiability checker.

4.4 Implementation

To show that our framework captures the essence of compositional symbolic execution, we have implemented one instantiation of the framework for the intermediate language of the .NET platform [15]. For bytecode manipulation, we use the Mono.Cecil [16] library and as constraint solver we use Z3 [17]. Our implementation achieves similar speedups as other compositional symbolic execution tools [12,11].

5 Properties

In this section, we show that the algorithm of Section 4 is precise, and we show that it is also progressing as long as the choices are fair. We only give a rough outline of the proof, since a more detailed exposition does not fit in the page limits. More details can be found in a technical report [14].

First, we show that compositional symbolic execution is precise.

Theorem 1 (Precision). *The algorithm is precise.*

Proof. To show precision, it suffices to show by induction over $\Rightarrow^*$ that the following properties are satisfied for each reachable analysis state a:

- Each leaf node ϵ of the summary $sum_a(m)$ of each method m is precise.
- Each finished leaf node ϵ of the summary $sum_a(m)$ of each method m is returning from m.
- All invocations $inv \in invs_a$ are precise, and the invocation graph is cycle free.
- The interpretations are precise.
- $Check(a, m, pc)$ is precise for each method m and condition pc.
- Each leaf node ϵ in rs_a is precise, and $Check(a, m, pc_\epsilon)$ succeeds (where m is active method of the symbolic state of ϵ).

Then, we show that compositional symbolic execution is monotonous.

Theorem 2 (Monotonicity). *For each program pr, concrete state s, and analysis states a, a' such that $a \Rightarrow a'$, if $\vdash^a reach(pr, s)$ then $\vdash^{a'} reach(pr, s)$.*

Proof. Follows from the fact that (a) $\Rightarrow$ never removes reachable leaf nodes ($rs_a \subseteq rs_{a'}$). (b) $\Rightarrow$ never removes invocations ($invs_a \subseteq invs_{a'}$).

Since the search tree of the algorithm is potentially infinite, monotonicity is not sufficient to find all reachable states: The algorithm might get stuck exploring only a subspace of the program. Fortunately, this can not happen if the analysis is *fair*, i.e. if each unknown node is eventually chosen by the algorithm.

Definition 4 (Fairness). *An application strategy of the compositional symbolic execution algorithm is* fair *if and only if for any analysis state a such that $a^0_{pr} \Rightarrow^* a$, for any unknown node $\epsilon \in sum_a(m)$, the algorithm always eventually chooses $\langle m, \epsilon \rangle$.*

Next, the progress argument relies on the validity of the following properties in each reachable analysis state a (which can again be proven by induction over $\Rightarrow^*$):

- The summaries sum_a are total.
- $Check(a, m, pc)$ is restricted complete for any method m and constraint pc.

Finally, we show that compositional symbolic execution algorithm is progressing if it is fair. The proof shows a slightly stronger property, namely that there always eventually is an analysis state where all concrete states on the execution trace that reaches s are concluded reachable. This is essential since it gives a stronger induction hypothesis: we assume that all but the last concrete state s is concluded reachable and we show that the analysis always eventually reaches an analysis state where s is also concluded reachable. This hypothesis is necessary since a state might only be reachable from one invocation that has not yet been discovered, whereas its predecessor is already reachable based on another invocation. Together with totality of the summaries and restricted completeness of reachability checking, this allows a compact and intuitive proof for progress.

Theorem 3 (Progress). *If the compositional symbolic execution algorithm is fair, then it is progressing.*

Proof. By induction on $\rightarrow^*$.

Base step. If s is the initial state, then $\vdash^a reach(pr, s)$ holds after applying the only possible analysis step.

Induction step. If $s^0_{pr} \rightarrow^* s'$ and $s' \rightarrow^* s$ and there always eventually is a reachable analysis state a' such that all concrete states from s^0_{pr} to s' are concluded reachable in a', then we must show that there always eventually is a reachable analysis state a such that $\vdash^a reach(pr, s)$. If $\vdash^{a'} reach(pr, s)$ already holds, then the proof is trivial.

1. First, we show that there exists an unknown node $\epsilon \in sum_{a'}(m)$ such that $Check(a', m, pc_\epsilon) = true$ and s is a concretization of ϵ. This means that if we choose $\langle m, \epsilon \rangle$, then the state s will become reachable in the next analysis state. This follows from the fact that the summaries are total, and restricted completeness of check.
2. By fairness, there always eventually exists a reachable analysis state a'' such that $\langle m, \epsilon \rangle$ has not been chosen yet, and is chosen in the next analysis step. Since $\langle m, \epsilon \rangle$ has not been chosen, it must still be in the summary of m ($\epsilon \in sum_{a''}(m)$). Because invocations are never removed ($invs_a \subseteq invs_{a''}$), the method $Check$ is monotonous and $Check(a'', m, pc_\epsilon) = true$. Therefore, if $\langle m, \epsilon \rangle$ is chosen in $a'' \Rightarrow a$, then $\vdash^a reach(pr, s)$.

6 Related Work

Compositional symbolic execution was first introduced in the context of SMART [11], as an extension of the automatic testing tool DART [18]. The authors informally argue that SMART is sound and complete (as a bugfinder) relatively to DART. In addition, DART is always sound (precise) and it is complete when it terminates [18]. The precision proofs depends critically on the dynamic aspect of SMART and DART. This paper

only depends on the precision of the interpretation rules. When the interpretation rules are imprecise in SMART or DART, it either causes incompleteness or non-termination. In addition, the progress property is stronger than completeness upon termination.

With demand-driven compositional symbolic execution [12], the dependency on the inner-most first search order of SMART is lifted. To achieve this, function summaries are encoded in the SMT-solver. In addition, the algorithm allows the SMT solver to construct inputs that follow unexplored paths through some methods. Such inputs may not reach their actual target but they always explore some new part of the program. This may be useful to alleviate the imperfectness of SMT solvers. We did not incorporate this in our framework, but the results can easily be extended. The authors claim relative completeness (as a bugfinder), and termination for programs with finite amounts of paths. The progress property in this paper is less algorithm specific and therefore more clear. In addition, it lifts the need for a termination argument. In the absence of fairness, demand-driven compositional symbolic execution does not satisfy the stronger progress property.

Finally, the system SMASH [13] combines the aspect of compositional analysis with may-must alternation. SMASH significantly outperforms both may-only, must-only and non-compositional may-must analysis. The analysis in this paper is a must analysis. As part of the soundness argument, the authors show that the must analysis of SMASH is precise. In addition, they show that the may analysis of SMASH is sound. Unfortunately, the combination of a sound may analysis with a precise must analysis is not necessarily a semi-decision procedure.

7 Conclusion

This paper creates a formal framework for compositional symbolic execution, based on a small but powerful calculus. We have modeled compositional symbolic execution as a transition system and formalized the meaning of precision and progress. In addition, we have proven that the algorithm is precise, and makes progress if the choices are fair. Finally, we have shown preliminary results of an implementation of the algorithm that is precise and progressing, and hence is a semi-decision procedure.

References

1. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
2. Tillmann, N., de Halleux, J.: Pex-white box test generation for.net. In: Proc. of Tests and Proofs 2008, pp. 134–153. Springer, Berlin (2008)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: Proc. of CCS 2006, pp. 322–335 (2006)
4. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS. The Internet Society, SanDiego (2008)
5. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI project: Software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
6. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. SIGOPS Oper. Syst. Rev. 39(5), 133–147 (2005)

7. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D.: BitScope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University (March 2007)
8. Anand, S., Pasareanu, C.S., Visser, W.: JPF–SE: A symbolic execution extension to Java Pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
9. Molnar, D.A., Wagner, D.: Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report 2007-23, University of California Berkeley (February 2007)
10. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. of SIGSOFT 2008/FSE-16 (2008)
11. Godefroid, P.: Compositional dynamic test generation. In: Proc. of POPL 2007, pp. 47–54 (2007)
12. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
13. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: unleashing the power of alternation. SIGPLAN Not. 45(1), 43–56 (2010)
14. Vanoverberghe, D., Piessens, F.: Precise and progressing compositional symbolic execution: Extended version (2010), http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW582.abs.html
15. European Computer Machinery Association: Standard ECMA-335: Common Language Infrastructure. 4th edn. (June 2006)
16. Evain, J.: Cecil, http://www.mono-project.com/Cecil
17. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78800-3_24
18. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40(6), 213–223 (2005)