Demand-Driven Compositional Symbolic Execution

Saswat Anand*[1], Patrice Godefroid[2], and Nikolai Tillmann[2]

[1] Georgia Institute of Technology saswat@cc.gatech.edu
[2] Microsoft Research {pg,nikolait}@microsoft.com

Abstract. We discuss how to perform symbolic execution of large programs in a manner that is both compositional (hence more scalable) and demand-driven. Compositional symbolic execution means finding feasible interprocedural program paths by composing symbolic executions of feasible intraprocedural paths. By demand-driven, we mean that as few intraprocedural paths as possible are symbolically executed in order to form an interprocedural path leading to a specific target branch or statement of interest (like an assertion). A key originality of this work is that our demand-driven compositional interprocedural symbolic execution is performed entirely using first-order logic formulas solved with an off-the-shelf SMT (Satisfiability-Modulo-Theories) solver – no procedure in-lining or custom algorithm is required for the interprocedural part. This allows a uniform and elegant way of summarizing procedures at various levels of detail and of composing those using logic formulas.
This novel symbolic execution technique has been implemented for automatic test input generation in conjunction with Pex, a general automatic testing framework for .NET applications. Preliminary experimental results are enouraging. For instance, our tool was able to generate tests triggering assertion violations in programs with large numbers of program paths that were beyond the scope of non-compositional test generation.

1 Introduction

Given a sequential program P with input parameters $\overrightarrow{I}$, the *test generation problem* consists of generating automatically a set of input values to exercise as many program statements as possible. There are essentially two approaches to solve this problem. *Static* test generation [17, 24, 8] consists in analyzing the program P statically, using symbolic execution techniques to attempt to compute inputs to drive P along specific paths or branches, but without ever executing the program. In contrast, *dynamic test generation* [18, 13, 6] consists in executing the program, typically starting with some random inputs, while simultaneously performing a symbolic execution to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer program

* The work of this author was done mostly while visiting Microsoft Research.

executions along alternative program paths. Since dynamic test generation extends static test generation with additional runtime information, it can be more powerful [13, 12], and is therefore used as the basis of this work.

As recently pointed out [12], automatic test generation (whether static or dynamic) does not scale to large programs with many feasible program paths, *unless* test generation is performed *compositionally*. Inspired by interprocedural static analysis, compositional test generation consists in encoding test results of lower-level functions with test *summaries*, expressed using function-input preconditions and function-output postconditions, and then re-using those summaries when testing higher-level functions. In contrast with traditional interprocedural static analysis, the framework introduced in [12] involves detailed summaries where arbitrary function preconditions and postconditions are represented using logic formulas, and the interprocedural analysis (test generation) is performed using an automated theorem prover. A key component of this approach is thus *compositional symbolic execution*: how to find feasible interprocedural program paths by composing symbolic executions of feasible intraprocedural paths, represented as logic "summaries".

In this paper, we develop compositional symbolic execution further. We present a detailed formalization of how to generate first-order logic formulas with uninterpreted functions in order to represent arbitrary function summaries and allow compositional symbolic execution using a SMT (Satisfiability-Modulo-Theories) solver. Our formalization generalizes the one of [12] as it allows incomplete summaries to be expanded *lazily* on a *demand-driven* basis, instead of being expanded in the fixed "innermost-first" order described in [12]. With demand-driven symbolic execution, as few intraprocedural paths as possible are symbolically executed in order to form an interprocedural path leading to a specific target branch or statement of interest (like an assertion). This increased flexibility also allows test generation to adapt dynamically, as more statements get covered, in order to focus on those program statements that are still uncovered. In practice, real-life software applications are very complex, and allowing the search to be demand-driven is often the key to reach a specific target in a reasonable time. It is also useful for selective regression testing aimed at generating tests targeted to cover new code embedded in old one.

We have implemented demand-driven compositional symbolic execution for automatic test input generation in conjunction with Pex [22], a general automatic testing framework for .NET applications. Preliminary experimental results are enouraging. For instance, our prototype implementation was able to generate tests triggering assertion violations in programs with large numbers of program paths that were beyond the scope of non-compositional test generation.

2 Background

We assume we are given a sequential program P with input parameters $\overrightarrow{I}$. *Symbolic execution* of P means symbolically exploring the tree $\mathcal{T}$ of all computations the program exhibits when all possible value assignments to input parameters

are considered. For each *control path* ρ, that is, a sequence of control locations of the program, a *path constraint* ϕ_ρ is constructed that characterizes the input assignments for which the program executes along ρ. Each variable appearing in ϕ_ρ is thus a program input, while each constraint is expressed in some theory T that can be decided by a theorem prover (for instance, including linear arithmetic, bit-vector operations, etc.). All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths ρ for which ϕ_ρ is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to ϕ_ρ exactly characterize the inputs that drive the program through ρ. Assuming that the automated theorem prover[3] used to check the satisfiability of all formulas ϕ_ρ is sound and complete, this use of symbolic execution for programs with finitely many paths amounts to program verification.

In practice, symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about symbolically with good enough precision at a reasonable cost. Whenever precise symbolic execution is not possible during dynamic test generation, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution [13].

Systematically executing symbolically *all* feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in presence of loops with unbounded number of iterations. This *path explosion* can be alleviated by performing symbolic execution *compositionally* [12].

Let us assume the program P consists of a set of functions. In what follows, we use the generic term of *function* to denote any part of the program P whose observed behaviors are summarized; obviously, any other kinds of program fragments such as arbitrary program blocks or object methods can be treated as "functions" as done in this paper. To simplify the presentation, we assume that the functions in P do not perform recursive calls, and that all the executions of P terminate. (These assumptions do not prevent P from possibly having infinitely many executions paths, as is the case if P contains a loop whose number of iterations may depend on some unbounded input.)

In compositional symbolic execution [12], a function summary ϕ_f for a function f is defined as a formula of propositional logic whose propositions are constraints expressed in the given theory T. ϕ_f can be computed by successive iterations and defined as a disjunction of formulas ϕ_w of the form $\phi_w = pre_w \wedge post_w$, where pre_w is a conjunction of constraints on the inputs of f while $post_w$ is a conjunction of constraints on the outputs of f. ϕ_w can be computed from the path constraint corresponding to the execution path w as described later. An

[3] In this paper, we use the terms "automated theorem prover" and "constraint solver" interchangeably. Recently developed SMT solvers such as Z3 [9] do not only decide satisfiability of a formula, but can also compute a model, i.e., a satisfying assigment, which effectively turns such theorem provers into constraint solvers.

input to a function f is any value that can be read by f in some of its executions, while an output of f is any value written by f in some of its executions and later read by P after f returns. To simplify the presentation and without loss of generality, we assume in what follows that each function takes a fixed number of arguments as inputs and returns a single value. Each value returned by a function is treated as a fresh symbolic input to the calling context.

3 Motivating Example and Overview

In this paper, we develop compositional symbolic execution further. We present a detailed formalization of how to generate first-order logic formulas with uninterpreted functions in order to represent arbitrary function summaries and allow compositional symbolic execution using a SMT (Satisfiability-Modulo-Theories) solver. Our formalization generalizes the one of [12] as it allows incomplete summaries to be expanded *lazily* on a *demand-driven* basis. With demand-driven symbolic execution, as few intraprocedural paths as possible are symbolically executed in order to form an interprocedural path leading to a specific *target* branch or statement of interest (like an assertion). This increased flexibility also allows test generation to adapt dynamically, as more statements get covered, in order to focus on those program statements that are still uncovered. It is also useful for selective regression testing aimed at generating tests targeted to cover new code embedded in old one.

To illustrate the motivation for demand-driven compositional symbolic execution, consider the simple program in Fig. 1, which consists of a top-level function `testAbs` which calls another function `abs`. The intraprocedural execution tree of each function is shown in Fig. 2. Each node in the execution tree represents the execution of a program statement such that a path from the root of the tree to a leaf corresponds to an intraprocedural path. Each path can be identified by its leaf node. Edges in executions trees are labeled with constraints expressed in terms of the function inputs. The conjunction of constraints labeling the edges of a path represents its associated path constraint as defined earlier. For example, Fig. 2(a) shows the execution tree of function `abs`, shown in Fig. 1, after execution of `abs` with input `x=1`. In what follows, we call a node *dangling* if it represents a path that has not been exercised yet. For example, after executing the `abs` with input `x=1`, any path on which input is less than or equal to `0` is not exercised. In Fig. 2(a), the sole dangling node is denoted by a circle.

The demand-driven compositional symbolic execution we develop in this work has two key properties: given a specific target to cover, it tries to (1) explore as few paths as possible (called *lazy exploration*) and (2) avoid exploring paths that can be guaranteed not to cover the target (called *relevant exploration*). We now illustrate these two features.

Lazy Exploration Assume that we first run the program of Figure 1 by executing the function `testAbs` with `p`=1 and `q`=1. This first execution will exercise the then branch of the first conditional statement in `abs` (node 3), as

```
int abs(int x){
  if(x > 0) return x;
  else if(x == 0)
         return 100;
  else return -x;
}
```

```
int testAbs(int p, int q){
   int m = abs(p);
   int n = abs(q);
   if(m > n && p > 0)
        assert false; //target
}
```

Fig. 1. Example program

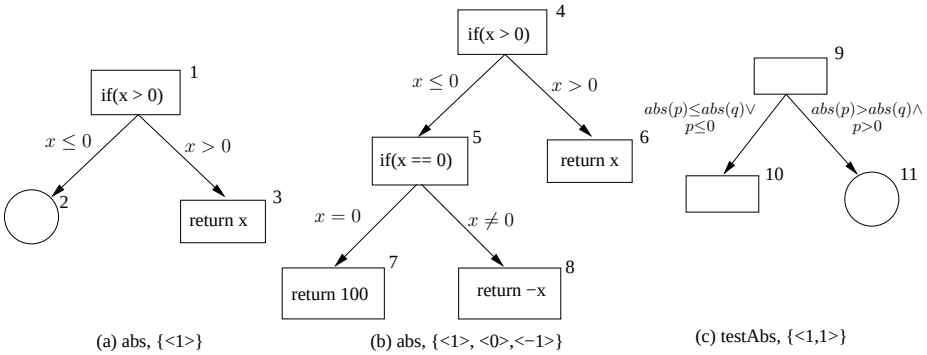

Fig. 2. Execution-trees of abs and testAbs functions from Fig. 1. Each execution tree represents paths exercised by a set of test-inputs, each of which is shown as a vector inside the curly braces.

well as the else branch of the conditional statement in `testAbs` (node 10). The execution trees of `abs` and `testAbs` resulting from this execution are shown in Fig. 2(a) and (c), respectively. Suppose we want to to generate a test input to cover node 11, corresponding to the assertion in `testAbs`. The search ordering described in [12] is not target-driven and would attempt to next exercise the unexplored paths in the innermost, lower-level function `abs`. In contrast, the more flexible formalization introduced in the next section allows us to check whether a combination of currently-known fully-explored intraprocedural paths are sufficient to generate a new test input covering the target node. In this example, this is the case as the assertion can be reached in `testAbs` without exploring new paths in `abs`, for instance with values `p`=2 and `q`=1.

Relevant Exploration Now, assume we first execute the program with inputs `p`=0 and `q`=1. Suppose our target is again node 11 corresponding to the assert statement. From the condition guarding the assert statement, observe that any combination of input values for `p` and `q`, where `p` has a non-positive value, has no chance to cover the target. As we will see, our proposed algorithm is able to infer such information automatically from the previous execution with inputs `p`=0 and `q`=1, and will thus prune automatically the entire sub-search tree where `p` is not greater than 0.

```
input  : Program P
output: A set of test inputs
exTrees ← emptyExTree ;
input ← RandomInput();
repeat
    if input ≠ emptyModel then
        exTrees ← Execute(P, input, exTrees);
        OutputTest(input);
    else
        RemoveDanglingNode(n);
    end
    n ← ChooseDanglingNode(exTrees);
    if n ≠ nil then
        input ← FindTestInput(exTrees, n);
    end
until n = nil ;
return exTrees ;
```

Algorithm 1: Test-input Generation algorithm

4 Demand-Driven Compositional Symbolic Execution

4.1 Main Algorithm

Algorithm 1 outlines our test-generation framework. Given a program P, Algorithm 1 iteratively computes a set of test inputs to cover all reachable statements in the program. The algorithm starts with an empty set of intraprocedural execution trees, and a random program input. It performs two steps in sequence until all reachable nodes in the program have been explored. (1) Function `Execute` executes the program with some test input, both normally and symbolically. During symbolic execution of the specific path exercised by the test input, new nodes and edges with constraint labels are added to the intraprocedural execution trees of the individual program functions being executed, while dangling nodes, used as place-holders along this specific path in previous executions, become regular nodes. (2) `ChooseDanglingNode` chooses a dangling node as the next target to be covered, using any heuristic (search strategy). If there is no dangling node remaining, the algorithm terminates. Otherwise, `FindTestInput` computes a test input to cover the target, as will be described next.

4.2 Compositional Symbolic Execution

In compositional symbolic execution, the condition under which a node in the execution tree can be reached from the program's entry point is the conjunction of (1) the condition under which the node's function can be reached, referred to as *calling context*; and (2) the condition under which the node can be reached within its function, referred to as the *local (intraprocedural) path constraint*.

Local (Intraprocedural) Path Constraint. The local path constraint of a node n in the intraprocedural execution tree $\mathcal{T}_f$ of function `f` is defined as the path constraint of the path w from the entry node of `f` to the statement represented by n. The local path constraint of node n, represented by $localpc(n)$, is expressed in terms of the input parameter symbols $\overrightarrow{\mathcal{P}_f}$ of `f` and represents a precondition $pre(w)$ for execution of the path w [12]. It is defined as follows.

$$localpc(n) := lpc_n \wedge \bigwedge_{\text{for each } g(\overrightarrow{a}) \text{ appearing in } lpc_n} D_g(\overrightarrow{a})$$

where lpc_n is the conjunction of constraints appearing on the edges of the path w from the root of $\mathcal{T}_f$ to n, and each definition predicate $D_g(\overrightarrow{a})$ represents the (possibly partial) summary currently available for function `g` called from `f`, with $\overrightarrow{a}$ as arguments. Definition predicates are formally defined as follows.

Definition predicate. When function `f` calls function `g` during symbolic execution, we treat the result of the function call to `g` as a (fresh) symbolic input to `f`. We represent the result by the expression $g(\overrightarrow{a})$, where $\overrightarrow{a}$ are the arguments expressed in terms of $\mathcal{P}_f$. If the result of the call is directly or indirectly used in a conditional statement of `f`, then $g(\overrightarrow{a})$ will appear in the path constraint. The function symbol g will be treated as an uninterpreted function symbol by the SMT solver, and we restrict possible interpretations by an axiom of the form $\forall x.\ g(x) = E[x]$, where $E[x]$ is an expression that may involve the bound variable x, or if `g` has a boolean type, $\forall x.\ g(x) \Leftrightarrow P[x]$, where $P[x]$ is a predicate over x. As an example, for the `abs` function in Fig. 1, abs can be defined as follows. ITE denotes the If-Then-Else construct.

$$\forall x.\ abs(x) \Leftrightarrow ITE(x > 0, x, ITE(x = 0, 100, -x))$$

However, return values of a function on some paths may be unknown since paths are explored on-demand. In those cases, we cannot use the above encoding directly. We could employ special *undefined* value that represents the result of an unexercised path, and lift all operations accordingly. Instead, our solution to this problem is use a *definition-predicate* D_g for each function symbol that represents the result of a method call. We define this predicate with the axiom δ_g as follows.

$$\delta_g :=\ \forall \overrightarrow{\mathcal{P}_g}.\ D_g(\overrightarrow{\mathcal{P}_g}) \Leftrightarrow \bigvee_{\text{leaf } l \text{ in } \mathcal{T}_g} localpc(l)\ \wedge\ ret(l)$$

where

$$ret(l) := \begin{cases} \mathcal{G}_l & \text{if } l \text{ is a dangling node} \\ g(\overrightarrow{\mathcal{P}_g}) = Ret_g(l) & \text{otherwise} \end{cases}$$

In the above definition, $Ret_g(l)$ represents the return-value of g, which is an expression in terms of $\overrightarrow{\mathcal{P}_g}$, on the fully-explored intraprocedural path represented by l. For each dangling node d, $\mathcal{G}_d$ represents an auxiliary boolean variable that uniquely corresponds to d; we use these boolean variables in Sec. 4.3 to control

the search by specifying whether the exploration of a new execution path through a dangling node is permissible.

For the example shown in Figure 1, suppose we execute `testAbs` with `p` = 1 and `q` = 1. Then the execution-trees for `abs` and `testAbs` built based on the path exercised by this input are shown in Fig. 2(a) and (c) respectively. Now, the local path-constraint of the node n, labeled 11 in the figure, will be as follows.

$$localpc(n) := abs(p) > abs(q) \wedge p > 0 \wedge D_{abs}(p) \wedge D_{abs}(q)$$

With the above input, since only the path where $x > 0$ has been explored in `abs`, there is a dangling node d , labeled 2, which represents the (unexplored) else branch of the conditional statement. The definition predicate D_{abs} is then defined by the following axiom.

$$\delta_{abs} := \forall x.\ D_{abs}(x) \Leftrightarrow ITE(x > 0, abs(x) = x, \mathcal{G}_d)$$

If all the paths of `abs` had been explored (as shown in Fig. 2(b)), its definition-predicate axiom would instead be as follows.

$$\begin{aligned}\delta_{abs} := \forall x.\ D_{abs}(x) \Leftrightarrow\ & (x \leq 0 \wedge x = 0 \wedge abs(x) = 100) \\ & \vee (x \leq 0 \wedge x \neq 0 \wedge abs(x) = -x) \\ & \vee (x > 0 \wedge abs(x) = x) \\ = \forall x.\ D_{abs}(x) \Leftrightarrow\ & ITE(x \leq 0, ITE(x = 0, abs(x) = 100, abs(x) = -x), abs(x) = x)\end{aligned}$$

Note that, with the specific innermost-first search order advocated in [12] for incrementally computing summaries, dangling and target nodes are always in the current innermost function in the call stack and the above formalization of partial summaries can therefore be simplified. In contrast, the formalization presented here is more general as it allows dangling nodes and target nodes to be located anywhere in the program.

Calling-context Predicate. The calling-context predicate associated with a function `f` describes under which conditions, and with which arguments, `f` can be reached. The calling-context predicate of function f, written as $C_f(\overrightarrow{a})$, evaluates to true iff on some program path f can be called with arguments $\overrightarrow{a}$. $C_f(\overrightarrow{a})$ is defined by the *calling-context axiom* γ_f as follows.

$$\gamma_f := \begin{cases} \forall \overrightarrow{a}.\ C_f(\overrightarrow{a}) \Leftrightarrow \overrightarrow{a} = \overrightarrow{I} & \text{if } f \text{ is the entry function of program } P \\ \forall \overrightarrow{a}.\ C_f(\overrightarrow{a}) \Leftrightarrow \bigvee_{\text{for each function } g \text{ in } P} C_f^g(\overrightarrow{a}) & \text{otherwise} \end{cases}$$

with

$$C_f^g(\overrightarrow{a}) :=\ \exists \overrightarrow{\mathcal{P}_g}.\ C_g(\overrightarrow{\mathcal{P}_g}) \wedge (knownC_f^g(\overrightarrow{a}) \vee unknownC^g)$$

where

$$\begin{aligned} knownC_f^g(\overrightarrow{a}) &:= \bigvee_{m \in callsites(\mathcal{T}_g, f)} \overrightarrow{a} = args(m) \wedge localpc(m) \\ unknownC^g &:= \bigvee_{\text{dangling node } d \text{ in } \mathcal{T}_g} localpc(d) \wedge \mathcal{G}_d \end{aligned}$$

We distinguish two cases in γ_f. First, if f is the entry function of the program P, then the arguments of f are the program inputs $\overrightarrow{I}$[4]. Otherwise, $C_f(\overrightarrow{a})$ is true iff f can be called from some function g (which may be f itself) with arguments $\overrightarrow{a}$; $C_f^g(\overrightarrow{a})$ represents the condition under which g may call f with arguments $\overrightarrow{a}$. $C_f^g(\overrightarrow{a})$ in turn evaluates to true iff (1) g itself can be called with arguments $\overrightarrow{\mathcal{P}_g}$; and either (2.a) f can be called from g in a *known* call site denoted by $m \in callsites(\mathcal{T}_g, f)$ with arguments $\overrightarrow{a} = args(m)$, where $args(m)$ denote the arguments (in terms of $\overrightarrow{\mathcal{P}_g}$) passed to call at m; or (2.b) f may be called (with unknown arguments) on a path in g, represented by a dangling node d, that has not been explored so far. For each of these known or possible call sites, the local path constraint $localpc(m)$ leading to the known call site m or $localpc(d)$ leading to a possible call site d, respectively, is appended as a condition necessary to reach the respective call site.

Consider again the program shown in Fig. 1 with `testAbs` as the top-level entry function. The calling-context predicate for `testAbs` is then defined by the following axiom.

$$\gamma_{testAbs} := \forall p, q.\ C_{testAbs}(p, q) \Leftrightarrow p = \overrightarrow{I}(0) \wedge q = \overrightarrow{I}(1).$$

For the function `abs`, the definition of the calling-context predicate is more complicated because `abs` can be called twice in `testAbs`. Suppose the execution trees of `abs` and `testAbs` are as shown in Fig. 2(b) and (c) respectively. For both known call-sites of `abs` in `testAbs`, where p and q are passed as arguments, *localpc* evaluates to *true*. And, there is one unknown call-site, which is represented by the dangling node d (labeled 11). For d, we have $localpc(d) := abs(p) > abs(q) \wedge p > 0 \wedge D_{abs}(p) \wedge D_{abs}(q)$. Now, $C_{abs}(a)$ is defined by the axiom γ_{abs} as follows.

$$\begin{aligned}
\gamma_{abs} &:= \forall a.\ C_{abs}(a) \Leftrightarrow C_{abs}^{testAbs}(a) \\
C_{abs}^{testAbs}(a) &:= \exists p, q.\ C_{testAbs}(p, q) \wedge (a = p \vee a = q \\
&\quad \vee (abs(p) > abs(q) \wedge p > 0 \wedge D_{abs}(p) \wedge D_{abs}(q) \wedge \mathcal{G}_d))
\end{aligned}$$

Note that an existential quantification is used in C_f^g to limit the scope of parameter symbols $\mathcal{P}_g$ to specific call-sites. However, this existential quantification can be eliminated by skolemization since it always appears within the scope of the universal quantifier in the definition of γ_f.

Also note that the formulation proposed in [12], does not include the notion of calling-context predicate because it dictates a fixed bottom-up ordering in which paths are explored. In this work, since we relax the restriction on the order so that paths can be explored in arbitrary order (e.g., on-demand), calling-context predicates become necessary.

Interprocedural path constraint. Given a node n in the intraprocedural execution tree $\mathcal{T}_f$ of a function f, the path constraints of multiple interprocedural paths to n are Ψ_n, which is defined as follows. Γ_{base} represents the set of

[4] W.l.o.g., we assume that the top-level entry function is not recursive (since one can always define a non-recursive higher-level wrapper function to call the original entry function).

basic axioms, e.g. length of an array is always non-negative or an axiom that encodes the transitivity property of sub-typing relation.

$$\Psi_n := localpc(n) \wedge C_f(\overrightarrow{\mathcal{P}_f}) \wedge \Gamma_{base} \wedge \bigwedge_{C_f(\overrightarrow{a}) \text{ appears in } \Psi_n} \gamma_f \wedge \bigwedge_{f(\overrightarrow{a}) \text{ appears in } \Psi_n} \delta_f$$

Ψ_n represents the disjunction of path-constraints of all interprocedural paths to target n that can be formed by joining intraprocedural paths, represented by execution-trees of different functions. An intraprocedural path p in $\mathcal{T}_f$ can be *joined* with an intraprocedural path q in $\mathcal{T}_g$, if (1) p ends at a leaf node (possibly a dangling node) in $\mathcal{T}_f$, and q starts at the target-node of an edge in $\mathcal{T}_g$ corresponding to a call-site of f in g; or, (2) p ends at the source-node of an edge representing a call-site of g in f and q starts at the entry-node of $\mathcal{T}_g$; or, (3) p ends at a dangling node, and q starts from the entry-node of $\mathcal{T}_g$, where g is any arbitrary function.

With compositional symbolic execution, the size of the path-constraints are linear in the sum of the sizes of the execution trees $\mathcal{T}_f$ [12].

Examples. As our first example, suppose the execution trees for `abs` and `testAbs` are as shown in Fig. 2(b) and (c), respectively. If the target is the node labeled 11, then the interprocedural path-constraint is as follows.

$$abs(p) > abs(q) \wedge p > 0 \wedge D_{abs}(p) \wedge D_{abs}(q) \wedge p > 0 \wedge C_{testAbs}(p, q)$$
$$\bigwedge \forall x.\ D_{abs}(x) \Leftrightarrow ITE(x \leq 0, ITE(x = 0, abs(x) = 100, abs(x) = -x), abs(x) = x)$$
$$\bigwedge \forall p, q.\ C_{testAbs}(p, q) \Leftrightarrow p = \overrightarrow{I}(0) \wedge q = \overrightarrow{I}(1)$$

As another example, suppose the execution tree for `abs` and `testAbs` are as shown in Fig. 2(b) and (c), respectively. Now if the target is node labeled 2, the path-constraint is as follows. $\mathcal{G}_{11}$ represents the unique boolean variable corresponding to the dangling node labeled 11.

$$x \leq 0 \wedge C_{abs}(x)$$
$$\bigwedge \forall x.\ D_{abs}(x) \Leftrightarrow ITE(x \leq 0, ITE(x = 0, abs(x) = 100, abs(x) = -x), abs(x) = x)$$
$$\bigwedge \forall a.\ C_{abs}(a) \Leftrightarrow \exists p, q.\ C_{testAbs}(p, q) \wedge (a = p \vee a = q \vee (abs(p) > abs(q) \wedge p > 0 \wedge D_{abs}(p) \wedge D_{abs}(q) \wedge \mathcal{G}_{11}))$$
$$\bigwedge \forall p, q.\ C_{testAbs}(p, q) \Leftrightarrow p = \overrightarrow{I}(0) \wedge q = \overrightarrow{I}(1)$$

4.3 Demand-Driven Symbolic Execution

In compositional symbolic execution, interprocedural paths are formed by combining intraprocedural paths. To allow compositional symbolic execution to be demand-driven, we allow in this work (unlike [12]) interprocedural paths to be formed by combining intraprocedural paths that end in dangling nodes. We call *partially-explored* an interprocedural path that goes through one or more dangling nodes; otherwise the path is called *fully-explored*. Note that a fully-explored path may end at, but not go through, a dangling node.

Algorithm 2 is used to find a feasible, fully-explored, interprocedural path form entry of the program to the target node using demand-driven compositional

symbolic execution. The algorithm corresponds to the subroutine `FindTestInput` function in Algorithm 1. The algorithm takes in a set of intraprocedural execution trees exTrees, and a dangling node n in one of these execution trees, which is the target to cover. It returns either (1) a designated value emptyModel representing the fact that the target node is unreachable, or (2) program inputs $\overrightarrow{I}$ that exercises a path that may cover the target. The algorithm uses an auxiliary function `FindTestInput`(Ψ), which returns a model for path constraint Ψ if it is satisfiable, or returns emptyModel otherwise. $G(\Psi)$ represents the set of all boolean flags that appear in the path constraint Ψ, each of which uniquely corresponds to a dangling node in exTrees. The algorithm first computes the interprocedural path constraint for the target node n in exTrees as presented in Sec. 4.2. Then it performs two steps, referred to as *lazy exploration* and *relevant exploration* in what follows.

Lazy Exploration In this step, the algorithm checks if it is possible to form a feasible, fully-explored, interprocedural path to n by combining only (fully-explored) intraprocedural paths in exTrees. To do so, it computes a constraint that represents the disjunction of path constraints of all such paths and checks its satisfiability. The new constraint is formed by conjoining Ψ_n with equations that set all variables but G_n in $G(\Psi_n)$ to $false$ so that all intraprocedural paths that end at a dangling node other than n are made infeasible. If the augmented constraint is satisfiable, `FindModel` returns a program test input that is guaranteed to cover the target (provided symbolic execution has perfect precision). Otherwise, we need to explore new partially-explored intraprocedural paths, which is done in the next step.

Relevant Exploration We say that a partially-explored, interprocedural path is *relevant* if it ends at the target. In other words, such a path starts at the program entry, goes through one or more dangling nodes, finally taking the path from the root node of $\mathcal{T}_f$ to the target node n, where $\mathcal{T}_f$ represents the execution tree of function f where n is located. In this second step, the algorithm checks if a feasible relevant path can be formed by combining all (both fully-explored and partially-explored) intraprocedural paths in exTrees. To do so, the algorithm checks satisfiability of Ψ_n with a second call to `FindModel`. If Ψ_n is unsatisfiable, the algorithm returns emptyModel representing unreachability of the target. Otherwise, it returns a program input that *may* exercise a path to the target. This time, the boolean variables in $G(\Psi_n)$ are not constrained to any specific value as is done in the previous step. As a result, the constraint solver assigns *true* to a boolean variable if the path to the corresponding dangling node is used to form the interprocedural path to the target. Such a relevant path is not guaranteed to reach the target, since the program's behavior at dangling nodes, which may appear on a relevant path, is currently unknown.

The following theorems define the correctness of the above algorithms. These theorems hold *assuming symbolic execution has perfect precision.* (Proof sketches are included in the appendix.)

Theorem (Relative Completeness) If Algorithm 2 returns emptyModel, then

input : Set of execution-trees exTrees, target node n to be covered
output: Program inputs that *may* cover n, or emptyModel if the target is unreachable

$\Psi_n \leftarrow$ `InterprocPC(n,exTrees);`
input $\leftarrow$ `FindModel(`$\Psi_n \wedge \bigwedge_{\mathcal{G}_d \in G(\Psi_n) \wedge d \neq n} \mathcal{G}_d = false$`);`
if input = emptyModel **then**
    input $\leftarrow$ `FindModel(`Ψ_n`);`
end
return input ;

Algorithm 2: Demand-driven, compositional FindTestInput algorithm

the target n is unreachable.

Theorem (Progress) If Algorithm 2 returns a program input $\overrightarrow{I}$ (different from emptyModel), then the execution of the program with $\overrightarrow{I}$ exercises a new path (i.e., at least one dangling node is removed from exTrees).

Theorem (Termination) If the program has a finite number of paths, Algorithm 1 terminates.

5 Pex

We implemented a prototype of the proposed technique in Pex [22], a general automatic testing framework for .NET programs. Pex generates test inputs for parameterized unit tests [23] using a variation of dynamic[13] test generation with Z3[9] as its constraint solver.

Background. Pex' goal is to analyze as many feasible execution paths of a given .NET program as possible in a given amount of time. Since dynamic symbolic execution discovers reachable statements of the program incrementally by diverting from already discovered execution paths, we want to reach particular target statements which are located just beyond the already explored frontier. *Dynamic* symbolic execution instead of static symbolic execution was implemented in Pex since the dynamic variant can be applied on real-world programs that interact with an external environment. In such cases, Pex simply does not see conditional branches performed in code not written in .NET, e.g. native code. The resulting constraint systems may no longer accurately characterize the program's behavior, and Pex prunes such paths. As a consequence, Pex always maintains an under-approximation of the program's behavior, which is appropriate for testing.

Choosing dangling nodes as targets. From the execution tree that has been explored at any point in time, Pex picks a dangling node as the next target. Pex implements a fair choice between all dangling nodes. Pex includes various fair strategies which partition all dangling nodes into equivalence classes, and

then pick a representative of the least often chosen class. The equivalence classes cluster nodes by mapping them (1) to the branch statement in the program of which the node is an instance, or (2) the stack trace at the time the node was created, or (3) the depths of the node in the execution tree. All such fair strategies are combined in a meta-strategy that performs a fair choice between the strategies.

Constraint solving. Pex performs various preprocessing steps to reduce the size of the formula before handing it over to the constraint solver, similar to constraint caching, and independent constraint optimization, as described in [6]. Pex employs Z3 [9] as its constraint solver. Except floating point arithmetic, Pex faithfully encodes all constraints arising in safe .NET programs such that Z3 can decide them with its built-in decision procedures for propositional logic, fixed sized bit-vectors, tuples, maps, and quantifiers. Besides the axioms presented in this paper, Pex also builds a background axiom that encodes the constraints of the .NET type system and virtual method dispatch as universally quantified formulas.

Applications. Pex has been applied within Microsoft to several .NET applications. In particular, it has found several known and unknown bugs, including security critical bugs, in released and still developed portions of the .NET Base Class Libraries.

6 Preliminary Experiments

We present experiments with three programs written in C# using both non-compositional and demand-driven compositional symbolic execution. These experiments were conducted on a 3.4 GHz machine with 2 GB memory.

HWM is program that takes a string as an argument, and an assertion fails if the input string contains all of the four substrings: “Hello”, “world”, “at”, “Microsoft!”. Although it is a simple program, it has hundreds of millions of feasible whole-program paths. The program has a main method that calls `contains(s,t)` four times in succession. `contains(s,t)` checks if string `s` contains substring `t`. `contains(s,t)` calls `containsAt(s,i,t)` that checks if `s` contains `t` starting from index `i` in `s`.

Parser is a parser for a subset of a pascal-like language. The program takes a string as argument, and successfully parses it if it represents a syntactically valid program in the language. An assertion is violated if parsing is successful. A valid program starts with the keyword “program” followed by an arbitrary string representing program name. Furthermore, the body of the program starts with keyword “begin” and end with keyword “end”. And the body may optionally include function definitions.

IncDec is a program that takes an integer as argument. It increments it several times and then decrements until a certain condition specified as an assertion is satisfied.

The table in Fig. 3 presents the results of those experiments. The three first columns represent the total number of executions, the total time taken over all

Benchmark	No. of Executions		Time in sec		time per execution		Exception found	
	new	old	new	old	new	old	new	old
HWM	37	maxed	65	705	1.75	0.02	yes	no
Parser	144	maxed	71	338	0.49	0.01	yes	yes
IncDec	74	1207	14	43	0.18	0.03	yes	yes

Fig. 3. Comparison between new (demand-driven, compositional) and old (non-compositional) symbolic execution techniques

executions, and time taken per execution, respectively. The last column shows whether the respective technique was able to generate an input that violates the assert statement contained in each program. In the column showing the number of executions, "maxed" denotes that non-compositional symbolic execution hits the upper bound on the number of execution of 20,000; in those cases, total execution time time represents the time taken to reach the upper bound.

We make the following observations from the table in Fig. 3. (1) The number of executions required with demand-driven compositional symbolic execution is at least several orders of magnitude smaller compared to non-compositional symbolic execution. (2) The improvement in total time cannot be measured as non-compositional symbolic execution technique hits the upper bound on the number of execution in two of the three cases. However, comparing the number of execution required for the two techniques, it will be safe to claim that the proposed technique can be many orders of magnitude faster than the other technique (if at all it finishes). (3) The time taken for each execution increases when the symbolic execution is demand-driven and compositional, as the formulas generated are more complicated and the constraint solver needs more time to solve those, although most can be solved in seconds. (4) In HWM, only the search with demand-driven compositional symbolic execution is able to find the assertion violation, whereas the non-compositional search is lost in search-space due to path explosion. The other two examples have fewer execution paths, and the search heuristics implemented in Pex are able to find the assertion violations, even with non-compositional searches.

7 Other Related Work

Interprocedural *static* analysis always involves some form of summarization [21]. Summaries are usually defined either at some fixed-level of abstraction, e.g., for points-to analysis [19], or as abstractions of intraprocedural pre and postconditions, e.g., projections onto a set of predicates [3, 25]. Even when a SAT solver is used for a precise intraprocedural analysis [7, 25, 2], the interprocedural part of the analysis itself is carried out either using some custom fixpoint computation algorithm [5, 25] or by (eagerly) in-lining functions [7, 2], the latter leading to combinatorial explosion.

In contrast with prior work on interprocedural static analysis, we represent function summaries as uninterpreted functions with arbitrary pre/postconditions represented as logic formulas, and we use an SMT solver to carry out the interprocedural part of the analysis. Of course, the SMT solver may need to in-line summaries during its search for a model satisfying a whole-program path constraint, but it will do so lazily, only if necessary, and while memoizing new induced facts in order to avoid re-inferring those later, hence simulating the effect of caching previously-considered calling contexts and new summaries inferred by transitivity, as in compositional algorithms for hierarchical finite-state machine verification [1].

How to perform *abstract* symbolic execution with simplified summary representations [16, 2, 14] in *static* program analysis is orthogonal to the demand-driven and compositionality issues addressed in our paper.

The use of automatically-generated software stubs [13] for abstracting (over-approximating) lower-level functions during dynamic test generation [20, 10] is also mostly orthogonal to our approach. However, the practicality of this idea is questionable because anticipating side-effects of stubbed functions accurately is problematic. In contrast, our approach is compositional while being grounded in testing and concrete execution,, thus without ever generating false alarms.

Demand-driven dynamic test generation for single procedures has previously been discussed in [18, 15]. This prior work is based on dataflow analysis, does not use logic and automated theorem proving, and does not discuss interprocedural analysis. As discussed earlier, our work extends the compositional test generation framework introduced in [12] by precisely formalizing how to implement it using first-order logic formulas with uninterpreted functions and a SMT solver, and by allowing it to be demand-driven.

8 Conclusion

This paper presents an automatic and efficient technique for test-input generation, which is both demand-driven and compositional. By demand-driven, we mean, given a target to cover, the technique aims to explore as few paths as possible (called lazy exploration), and avoid exploring paths that can be guaranteed not to cover the target (called relevant exploration). By compositional, we mean, instead of exploring each interprocedural path in isolation, the technique finds feasible, interprocedural paths by combining intraprocedural paths. Because the technique is demand-driven it is very efficient when the goal is to cover a particular location in the program (e.g., assertions). And, due to its compositional feature, it can alleviate the path-explosion problem, which severely limits the scalability of automatic test-input generation techniques. We have a prototype implementation of the proposed technique in Microsoft's Pex test-generation tool. Preliminary experimental results are promising. Currently, we are extending our prototype to handle implementation issues such as summarizing side-effects through the heap. Future work includes applying the technique to a larger set of programs to further assess its effectiveness.

9 Acknowledgments

We would like to thank Jonathan 'Peli' de Halleux, one of the main developers of Pex, for his help. We would also like to thank Nikolaj Bjorner and Leonardo de Moura for the Z3 theorem prover, and their support while we were using early versions of Z3.

References

1. R. Alur and M. Yannakakis. Model Checking of Hierarchical State Machines. In *Proceedings of FSE'98*, pages 175–188, 1998.
2. D. Babic and A. J. Hu. Structural Abstraction of Software Verification Conditions. In *CAV'2007*, Berlin, July 2007.
3. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of PLDI'2001*, 2001.
4. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45, New York, NY, USA, 1990. ACM Press.
5. W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.
6. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.
7. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of TACAS'2004*, 2004.
8. C. Csallner and Y. Smaragdakis. Check'n Crash: Combining Static Checking and Testing. In *ICSE'2005*. ACM, May 2005.
9. L. de Moura and N. Bjorner. Z3, 2007. Web page: `http://research.microsoft.com/projects/Z3`.
10. D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of ISSTA'2007*, 2007.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'2002*, volume 37-5, pages 234–245, June 2002.
12. P. Godefroid. Compositional Dynamic Test Generation. In *POPL'2007*, pages 47–54, Nice, January 2007.
13. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*, pages 213–223, Chicago, June 2005.
14. D. Gopan and T. Reps. Low-level Library Analysis and Summarization. In *CAV'2007*, Berlin, July 2007.
15. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *Proceedings of ASE'2000*, pages 219–227, September 2000.
16. S. Khurshid and Y. L. Suen. Generalizing Symbolic Execution to Library Classes. In *PASTE'2005*, Lisbon, September 2005.
17. J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.
18. B. Korel. A Dynamic Approach of Test Data Generation. In *ICSM*, pages 311–317, San Diego, November 1990.
19. V. B. Livshits and M. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of FSE'2003*, 2003.

20. R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical report, UC Berkeley, 2007.
21. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of POPL'95*, pages 49–61, 1995.
22. N. Tillmann and J. de Halleux. Pex, 2007. Web page: `http://research.microsoft.com/Pex`.
23. N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC-FSE,*, pages 253–262. ACM, 2005.
24. W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *ISSTA'04*, Boston, July 2004.
25. Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of POPL'2005*, 2005.

A Proofs of Theorems

Theorem (Relative Completeness) If Algorithm 2 returns emptyModel, then the target n is unreachable.

Proof by contradiction: Assume n is reachable, i.e., there is feasible, interprocedural path p to the target n. We can split p into a set of intraprocedural subpaths $SP := \{p_1, \ldots, p_n\}$. Let SP' be the subset SP' of SP containing all the subpaths in p that have already been fully-explored in the current set of execution trees exTrees available during the search. If $SP = SP'$, each subpath in SP is in one of the execution trees exTrees, and therefore the lazy-exploration step will find an input to cover the target (assuming symbolic execution has perfect precision). Otherwise, consider the first subpath w along p that is not in SP'. There must be a dangling node in exTrees representing a path that matches a prefix of the subpath w. By construction, this partially-explored matching path can be combined with other subpaths in SP' to form a partially-explored, interprocedural path q to the target. Furthermore, since p extends q, the path constraint of q is logically implied by that of p. Since p is feasible (hence its path constraint is satisfiable), q must also be feasible, thus a non-empty model will be returned in the relevant-exploration step of the algorithm.

Theorem (Progress) If Algorithm 2 returns a program input $\overrightarrow{I}$ (different from emptyModel), then the execution of the program with $\overrightarrow{I}$ exercises a new path (i.e., at least one dangling node is removed from exTrees).

Proof sketch. Two cases are possible: either $\overrightarrow{I}$ is computed during the lazy-exploration step, or during the relevant-exploration step. If $\overrightarrow{I}$ is found during the lazy-exploration step, then the path taken by $\overrightarrow{I}$ is guaranteed to take a path through the target node (assuming symbolic execution has perfect precision), which is a dangling node. For the second case, let us assume the contrary: $\overrightarrow{I}$ exercises a path that does not go through any dangling node. In that case, by construction of the formula Ψ_n, the path must end at the target node n (assuming again symbolic execution has perfect precision). Since it does not go through

any dangling node, it must have been discovered in the lazy-exploration step. A contradiction.

Theorem (Termination) If the program has a finite number of paths, Algorithm 1 terminates.

Proof. Immediate since every call to `FindTestInput` (Algorithm 2) from Algorithm 1 results in the exploration of at least one new intraprocedural path (see previous theorem), and the number of such paths is assumed to be finite.

B Details of the Pex Test-Generation Tool

Monitoring. Pex monitors the execution of a .NET program through code instrumentation. Pex plugs into the .NET profiling API and rewrites the instruction sequence of a method just before the intermediate language is translated into machine code. The instrumented code drives a "shadow interpreter" in parallel to the actual program execution. Basically, there is one callback to per instruction. The "shadow interpreter"

- constructs symbolic representations of the executed operations over logical variables instead of the concrete program inputs;
- maintains and evolves a symbolic representation of the entire program's state at any point in time;
- records the conditions over which the program branches.

Pex' interpreter models the behavior of all verifiable .NET instructions precisely, and models most unverifiable (involving unsafe memory accesses) instructions as well.

Symbolic state representation. A symbolic program state is a predicate over logical variables together with an assignment of expressions over logical variables to locations, just as a concrete program state is an assignment of values to locations. The locations of a state may be static fields, instance fields, method arguments, locals, and positions on the operand stack.

Pex' expression constructors include primitive constants for all basic .NET data types (integers, floating point numbers, object references), and functions over those basic types representing particular machine instructions, e.g. addition and multiplication. Pex uses tuples to repesent .NET value types ("structs"), and map data types to represent instance fields and arrays, similar to the heap encoding of ESC/Java [11].

Pex implements various techniques to reduce the enormous overhead of the symbolic state representation. Before building a new expression, Pex always applies a set of reduction rules which compute a normal form. A simple example of a reduction rule is constant folding, e.g. $1 + 1$ is reduced to 2. All logical connectives are transformed into a BDD representation with if-then-else expressions [4]. All expressions are hash-consed, i.e. only one instance is ever allocated in memory for all structurally equivalent expressions.

Based on the already accumulated path condition, expressions are further simplified. For example, if the path condition already established that $x > 0$, then $x < 0$ simplifies to `false`.