SPA: Symbolic Program Approximation for Scalable Path-sensitive Analysis

Raul Santelices and Mary Jean Harrold
College of Computing, Georgia Institute of Technology
E-mail: {raul|harrold}@cc.gatech.edu

ABSTRACT

Symbolic execution is a static-analysis technique that has been used for applications such as test-input generation and change analysis. Symbolic execution's path sensitivity makes scaling it difficult. Despite recent advances that reduce the number of paths to explore, the scalability problem remains. Moreover, there are applications that require the analysis of all paths in a program fragment, which exacerbate the scalability problem. In this paper, we present a new technique, called Symbolic Program Approximation (SPA), that performs an approximation of the symbolic execution of all paths between two program points by abstracting away certain symbolic subterms to make the symbolic analysis practical, at the cost of some precision. We discuss several applications of SPA, including testing of software changes and static invariant discovery. We also present a tool that implements SPA and an empirical evaluation on change analysis and testing that shows the applicability, effectiveness, and potential of our technique.

1. INTRODUCTION

Symbolic execution [11, 28] is a well-known program analysis technique used in software-engineering tasks such as test-case generation (e.g., [24]) and test-suite augmentation (e.g., [31]). Symbolic execution analyzes program paths one by one with full precision. For each path, symbolic execution computes the values of program variables and the conditions (constraints) required to reach program points along that path as expressions in terms of input variables, which are represented by symbols. Unfortunately, techniques based on symbolic execution do not scale well because of the number of paths in a program is typically infinite or grows exponentially with the size of the program (a problem known as *path explosion*). The problem is exacerbated by the presence of library calls and the difficulty of solving certain types of constraints.

Some applications, such as change analysis [2, 29, 31] for testing and maintenance, require the symbolic execution of all paths between two program points. For that purpose, Apiwattanapong and colleagues presented a practical variant of symbolic execution, called *partial symbolic execution* [2, 31], that starts at a change in the program instead of the entry of the program and explores only paths up to a certain distance from the change. The test-suite augmentation technique based on this variant finds the precise conditions under which a change affects the state of the program and propagates those effects along all paths up to the specified distance (the goal of which is to ultimately affect the output). However, such techniques also suffer from the path explosion problem, so the distances they can be reach in practice are limited.

Recently, novel approaches for symbolic execution of individual paths have been presented that heuristically select the next path to explore in a way that gets increasingly "closer" to a coverage goal (program point) and thus, reduces the number of paths analyzed [24, 32, 38]. Further research on modularity has made symbolic execution more efficient by computing intermediate summaries that help reduce the number of paths to explore [1, 23]. However, despite these advances, the scalability of symbolic execution is still limited by the path-explosion problem. For relatively large programs and components, only a small set of paths can be symbolically executed within a certain time and memory budget.

A different, orthogonal direction has been taken by researchers in model-checking [10], who have defined diverse abstractions that significantly reduce the state space to explore [3, 4, 7, 13]. Because the goal of model-checking is to verify or show violations of properties, if the abstraction is too coarse, researchers have proposed to iteratively refine the abstraction using information from infeasible counterexamples [5, 9, 25]. However, this process can lead to an explosion in the state space and fail to produce an answer.

In this paper, we present *Symbolic Program Approximation* (SPA), a new, general, and scalable approach to symbolic execution on all program paths between two points. SPA addresses the path explosion problem by analyzing program paths in groups rather than individually, taking advantage of the control- and data-dependence structure of the program, and approximating path groups in a way that trades off precision for scalability. In this way, approximate results for symbolic execution can be obtained for larger programs than traditional symbolic execution can handle. The SPA technique is based on two main insights. First, multiple paths can be combined into sets of related paths, called *path families*, such that the symbolic conditions for covering a path family (i.e., covering any path in that family) are simpler than the conditions for covering individual paths in that family. Second, symbolic conditions can be arbitrarily dropped in a safe manner to produce an abstract interpretation [14] (i.e., an overapproximation) of the results of symbolic execution on path families. SPA is a parameterized algorithm for approximate symbolic execution that can be instantiated by partition-abstraction strategies designed specifically for the task at hand (e.g., change analysis). An effective strategy distributes all program paths between two points into a set of path families that is small enough to symbolically execute in practice and that is also effective for a given task. For example, an effective set of path families for testing a change specifies (partially complete) symbolic conditions for the propagation of the effects of the change along those path families; satisfying such conditions for a path family makes it likely that the effects of the change will propagate along some path in that family and affect the output.

Our technique uses three mechanisms to compute path families and perform abstractions on demand. First, SPA defines path families in terms of the interprocedural control-dependence graph [34] instead of the control-flow graph, which is the representation traditionally used in symbolic execution. Such a path family concisely describes all control-flow paths that follow the same control-dependence edges. The edges in a path family correspond to the terms in the *path-family condition* (PFC)—the generalization of a path condition to multiple paths—computed by SPA for that family. Second, SPA uses the interprocedural data dependencies in the program to compute symbolic values and partition path families by directly traversing such dependencies and identifying the intermediate PFCs for those dependencies. Because this process is performed on demand, many combinations of intermediate conditions do not need to be analyzed, which further reduces the total number of paths with respect to traditional symbolic execution. Finally, SPA uses abstract interpretation [14] to avoid partitioning some path families, overapproximating their symbolic execution instead. This abstraction step makes symbolic execution practical at the cost of some precision. The result safely approximates PFCs and symbolic values of variables.

SPA has a number of applications in program analysis and testing. For example, the results of SPA can be used in software testing to approximate the behavioral differences between two versions of a program; these differences can be used to guide test-site augmentation [2,31]. For another example, these results can be used to approach test-case generation [19,24,32] from an all-paths perspective. SPA can also be used in other software-engineering tasks such as invariant discovery [15,18,22,35], bug finding [3], and modular analysis: SPA computes symbolic expressions that model the effects of a program module in terms of its input; such expressions are invariants that overapproximate the behavior of the module and can be checked against a specification. In addition, the abstraction resulting from our technique can be refined iteratively to produce increasingly more precise abstractions.

In this paper, we also present the SPA tool that implements our technique for Java programs. SPA uses models of library calls to focus the analysis effort on the application. We instantiated the SPA tool with two strategies. The first strategy analyzes path families covering sequences of program dependencies from a change point to a certain dependence-distance d, without abstractions. This instantiation of SPA corresponds to the technique called *partial symbolic execution* [2,31]. The second strategy also analyzes all paths in dependence sequences from a change point, but uses a limit e for the depth of the resulting symbolic expression tree, abstracting all sub-expressions below e; using this strategy, and for the same computational budget, SPA can analyze more paths—with some imprecision—than partial symbolic execution, including parts of the program that might contain important information that is missed by a fully precise analysis. Using the SPA tool, we performed an empirical study of these two instantiations for change analysis and test-suite augmentation. The results of our study show that the second strategy, using abstractions, is as effective or more effective for this application than the first strategy, while being considerably more efficient. Therefore, we show that the SPA framework can be instantiated to produce more effective and efficient analyses than techniques based on fully precise, traditional symbolic execution.

The main contributions of this paper are:

- The SPA technique for symbolic execution of all paths between two program points that leverages path families, data dependencies, and safe abstractions that trade precision for scalability.
- A set of applications of SPA for software engineering.
- A tool that implements our technique and empirical studies using this tool on Java bytecode programs that show the effectiveness of SPA for test-suite augmentation.

2. BACKGROUND

This section first introduces an example that we use throughout the paper, and then overviews the main concepts required for our technique: data and control dependencies, and symbolic execution.

2.1 Example

To discuss the background and our new technique, we use the example function `addElem`, given on the left in Figure 1. Function `addElem` takes as input unsigned integers `a`, `b`, and `c`, and map reference `m`, which associates integers with lists of integers. In statements 1 and 2, if the map is not initialized, `addElem` creates a new map, and statement 3 initializes variable `sz` to track the size of the map. In statements 4–10, the function retrieves the list `l` associated with key `a`, or, if no entry for key `a` exists, initializes that list to the single-element list `{a}`, adds entry `a`$\rightarrow$`l` to the map, and updates `sz` accordingly. Statements 6 and 7 also initialize `l` to `{a}` if the retrieved list is empty. In statements 11–14, the function adds `b` to a list `l` mapped to `a`, as long as that list is not longer than the size of the map plus a tolerance `c`, and prints `"succeeded"` or `"failed"` accordingly. The function finally returns the tracked map size.

2.2 Control and Data Dependencies

Informally, statement s_1 is *control-dependent* [20] on statement s_2 with label L if, in the control-flow graph (CFG)[1] for the procedure,[2] the node associated with s_2 has two or more outgoing edges, and for at least one (with label L) but not all of these edges, every path reaching that edge also reaches the node corresponding to s_1. Control dependencies can be represented in a *control-dependence graph* (CDG), in which nodes represent statements and edges represent the control-dependencies between the statements.

To illustrate, consider function `addElem`, shown in Figure 1. The control-flow graph (CFG) for the function is shown in the center in the figure. By inspecting the CFG, we can see that statement 6 executes only if statement 4 evaluates to `true` and that statement 7 executes only if statement 6 evaluates to `true`. Thus, statement 6 is control dependent on 4T and statement 7 is control dependent on 6T. In the CDG for `addElem` (shown on the right in Figure 1), there is an edge from node 4 to node 6 labeled "T" and an edge from node 6 to node 7 also labeled "T."

Statements in a procedure that are not control dependent on any other statement in that procedure, such as statements 1, 3, 4, 11, and 15 in `addElem`, are control dependent on the start of the procedure. The CDG on the right in Figure 1 shows this dependence on the node labeled `START`. Additionally, statements in a CDG that have the same control dependencies can be grouped into the same *control-dependence region* (cd-region) [20]. In the CDG in Figure 1, nodes in the same cd-region are shaded. For example, in Figure 1, nodes representing entry, exit, 1, 3, 4, 11, and 15 are in a cd-region controlled by `START`-T, and nodes representing statements 13 and 14 are in a cd-region controlled by 11F.

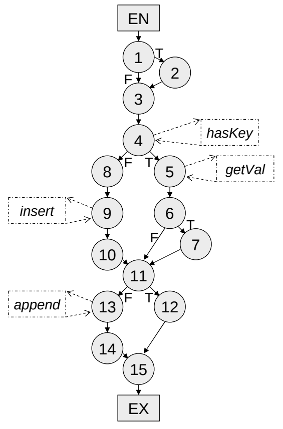
Nodes in a cd-region are not necessarily ordered by their execution order, but for our purposes, we require this order. The CDG in Figure 1 shows the nodes ordered from left to right within each cd-region. For example, in the cd-region controlled by 11F, the execution order of the nodes is 13, 14.

Control-dependencies can be computed for single procedures (intraprocedural), as shown in Figure 1, or across procedures (interprocedural). Informally, an *interprocedural control-dependence*

[1] In a *control-flow graph*, nodes represent statements, edges represent the flow of control between the statements, and special nodes, labeled EN and EX, represent entry to and exit from the procedure, respectively.

[2] We use procedure to represent a procedure, a method, a function, or a monolithic program.

```
uint addElem(uint a,b,c; Map m)
     List l, int sz
  1.  if m == null
  2.     m = new Map
  3.  sz = m.size
  4.  if hasKey(m, a)
  5.     l = getVal(m, a)
  6.     if l.length == 0
  7.        l = {a}
      else
  8.     l = {a}
  9.     m = insert(m, a→l)
 10.     sz++
 11.  if l.length >= sz + c
 12.     print "failed"
      else
 13.     l = append(l, b)
 14.     print "succeeded"
 15.  return sz
```



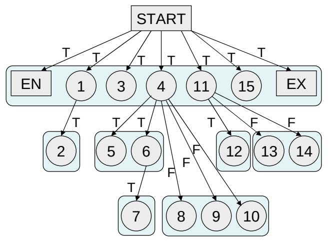


Figure 1: Example function `addElem` (left), its control-flow graph (CFG) (center), and its control-dependence graph (CDG) (right).

graph [34] (ICDG) contains one CDG for each procedure, edges that connect the CDGs at call and return sites, and additional artifacts that account for the calling context of called procedures.

Statement s_1 is *data dependent* on statement s_2 if (1) s_2 defines variable v, (2) there is a path from s_2 to s_1 that contains no redefinition of v, and (3) s_1 uses v. For example, in Figure 1, statement 11 is data dependent on statement 3 for variable sz, because statement 3 defines sz, statement 11 uses sz, and there is a definition-clear path (e.g., <3,4,5,6,11>) from statement 3 to statement 11.

2.3 Symbolic Execution

Symbolic execution [28] analyzes a program by executing it with symbolic inputs along some program path. Symbolically executing all paths in a program to a given point (if feasible) effectively describes the semantics of the program up to that point. Symbolic execution represents the values of program variables at any given point in a program path as algebraic expressions by interpreting the operations performed along that path on the symbolic inputs. The *symbolic state* of a program at a given point consists of the set of symbolic values for the variables in scope at that point. The set of constraints that the inputs must satisfy to follow a path is called a *path condition* (PC) and is a conjunction of constraints p_i or $\neg p_i$ (depending on the branch taken), one for each predicate traversed along the path. Each p_i is obtained by substituting the variables used in the corresponding predicate with their symbolic values. Symbolic execution on all paths to a program point represents the set of possible states at that point as a disjunction of clauses, one for each path that reaches the point. These clauses are of the form $PC_i \Rightarrow S_i$, where PC_i is the path condition for path i, and S_i is the symbolic state after executing path i.

To illustrate, consider path <EN,1,2,3,4,8> in Figure 1. We denote a symbolic input by the name of the input variable and a zero subscript. In this example, symbolic execution starts with the symbolic inputs $\mathtt{a}_0$, $\mathtt{b}_0$, $\mathtt{c}_0$, and $\mathtt{m}_0$ and uses these symbols as the initial values of the respective variables. Also, the path condition PC is initially empty. At statement 1 in the example path, there is no change in any variable, but when moving to statement 2, symbolic execution updates the path condition to $\mathtt{m}_0\mathtt{==null}$. At statement 2, variable `m` equals `new Map`. At statement 3, variable `sz` is initialized to `(new Map).size`. At branch 4F, symbolic execution updates the path condition to $\mathtt{m}_0\mathtt{==null} \wedge \neg\mathtt{hasKey(new\ Map,a}_0\mathtt{)}$. Finally, at statement 8, `sz` is initialized to `{a}`. Thus, at the end of path <EN,1,2,3,4,8>, symbolic execution obtains the following path condition and symbolic state:

PC = $\mathtt{m}_0\mathtt{==null} \wedge \neg\mathtt{hasKey(newMap,a}_0\mathtt{)}$
$\mathtt{a=a}_0$, $\mathtt{b=b}_0$, $\mathtt{c=c}_0$, `m=new Map`, `sz=(new Map).size`, `l={a}`.

3. THE SPA TECHNIQUE

This section presents the SPA technique for symbolic program approximation. We present an overview of our technique using the example in Figure 1 (Section 3.1) and then we describe the components of SPA (Sections 3.2 and 3.3).

3.1 Overview of the Technique

Consider Figure 1 and assume that we want to compute the conditions in this function for which adding an element to the list fails—that is, the conditions for all paths that reach statement 12. By inspecting the function, we can see that there are six paths in `addElem` that reach statement 12; the path condition for each path is some combination of truth values for the conditions at statements 1, 4, and 6, and the condition 11T (i.e., statement 11 evaluates to true). Unfortunately, there are in reality many more paths to statement 12 because these paths contain calls to `hasKey` and other functions. Function `hasKey`, for example, can have a large or infinite number of paths—a typical implementation of maps is hash maps, which are efficient but complex. Thus, the total number of paths can be unmanageable. Even worse, these auxiliary functions might reside in libraries whose code is not available. Because of these difficulties, traditional symbolic execution might have considerable trouble finding feasible paths whose constraints can be solved to find a satisfying input, let alone finding the conditions for all program paths that cover the goal.

To solve the problem illustrated in this example, our technique takes an opposite approach to traditional symbolic execution: instead of exploring paths one by one, SPA starts with one *path family* that groups all paths from the entry of `addElem` to the goal (statement 12), and computes a symbolic condition for that path family. We call this condition a *path-family condition* (PFC)—the multi-path version of a path condition. The path family for all paths from entry to statement 12 is described simply by the symbolic value of 11T. The conditions at statements 1, 4, and 6 are not required because, regardless of the values that these conditions take, the program execution will always reach statement 11. All paths that branch at statements 1, 4, and 6 (plus all paths inside auxiliary functions) are implicitly specified as part of this path family.

The problem of finding the PFC to the goal using SPA is thus reduced to finding the symbolic value of condition 11T: the length of the list associated with `a` is greater than or equal to the sum of `sz` and `c`. To determine the length of the list, for instance, it is necessary to examine more closely the original path family, including the auxiliary functions, and look for the definitions of `l` that reach 11: statements 5, 7, and 8, which assign to `l` the values `getVal(m,a)`, `{a}`, and `{a}`, respectively. To reach statement 11, these definitions require branch sequences <4T,6F>, <4T,6T>, and <4F>, respectively, which partition the original path family into three sub-path-families.

At this point, we notice that the definition of `l` at 5 depends on the return value of `getVal` and that conditions 4T and 4F depend on the return values of `hasKey`. Therefore, the analysis has two alternatives: (1) enter these functions and analyze their contents (a complex endeavor), or (2) apply a core mechanism of SPA that abstracts conditions 4T and 4F and variable `l` at 5 as $*$, which represents the *top* value (i.e., the set of all possible values for a given type).[3] In this example, we let SPA choose the second alternative. For example, the value $*$ for condition 4T means that this condition has a value set {`true`, `false`}; both elements in the set are valid values for this condition in our abstraction.

For the rest of this paper, $\texttt{v}_n$ denotes the use of variable `v` at statement n. The special value n = 0 denotes the entry of the program. Thus, with inputs denoted symbolically as $\texttt{a}_0$, $\texttt{b}_0$, $\texttt{c}_0$, and $\texttt{m}_0$, respectively, condition 11T is $\texttt{l}_{11}\texttt{.length}\geq\texttt{sz}_{11}\texttt{+c}_0$. (Note that `a`, `b`, and `c` are never modified, so any use of these variables can be safely replaced by inputs $\texttt{a}_0$, $\texttt{b}_0$, and $\texttt{c}_0$, respectively). Replacing $\texttt{l}_{11}$ by its definitions and their respective path conditions, we obtain the following case-style expression for 11T:

$$\begin{array}{l}\{\ \texttt{case 4T,6F: } *\texttt{.length} \geq \texttt{sz}_{11} + \texttt{c}_0\\ \quad\texttt{case 4T,6T: } \{\texttt{a}_0\}\texttt{.length} \geq \texttt{sz}_{11} + \texttt{c}_0\\ \quad\texttt{case 4F: } \{\texttt{a}_0\}\texttt{.length} \geq \texttt{sz}_{11} + \texttt{c}_0\ \}\end{array}$$

which, because $\{\texttt{a}_0\}$ is a list of one element, simplifies to

$$\begin{array}{l}\{\ \texttt{case 4T,6F: } *\texttt{.length} \geq \texttt{sz}_{11} + \texttt{c}_0\\ \quad\texttt{case 4T,6T: } 1 \geq \texttt{sz}_{11} + \texttt{c}_0\\ \quad\texttt{case 4F: } 1 \geq \texttt{sz}_{11} + \texttt{c}_0\ \}\end{array}$$

Use $\texttt{sz}_{11}$ has reaching definitions at statements 3 and 10 whose conditions to reach 11 are <4T> and <4F>, respectively. Thus, $\texttt{sz}_{11}$ = { $\texttt{case 4T: m}_3\texttt{.size; case 4F: m}_3\texttt{.size+1}$ }, where $\texttt{m}_3$ can be concisely expressed by $\texttt{(m}_0\texttt{ =null)?(new Map):m}_0$. Thus, $\texttt{m}_3\texttt{.size}$ can be simplified to $\texttt{(m}_0\texttt{ =null)?0:m}_0\texttt{.size}$. Now, because both 4T and 4F are $*$, use $\texttt{s}_{11}$ evaluates to $\{\texttt{case m}_0\texttt{ =null:\{0,1\}; case m}_0 \neq\texttt{null:\{m}_0\texttt{.size,m}_0\texttt{.size+1\}}\}$. Also, both conditions at 6 are $*$ because the expression $*\texttt{.length=0}$ is both true and false. Therefore, the final expression for 11T to reach our goal (statement 12) is the PFC

$$\begin{array}{l}\{\ *\texttt{.length} \geq \texttt{((m}_0\texttt{ =null)?0:m}_0\texttt{.size) + c}_0\texttt{,}\\ \quad 1 \geq \texttt{((m}_0\texttt{ =null)?0:m}_0\texttt{.size) + c}_0\texttt{,}\\ \quad 1 \geq \texttt{((m}_0\texttt{ =null)?0:m}_0\texttt{.size) + 1 + c}_0\ \}\end{array}$$

The resulting approximate PFC evaluates to true if any of its three values is true. In other words, the abstractions applied by SPA make all three values valid for <11T>, for any input.[4] An input generator that solves this PFC to execute statement 12 must pick an input that satisfies all three values of <11T> to ensure that the goal is reached. If the input generator knows that the fields `length` and `size` of non-null lists and maps are never negative, then the generator can strengthen this PFC to $\texttt{0}\geq\texttt{((m}_0\texttt{ =null)?0:m}_0\texttt{.size)}$ $\texttt{+c}_0$, and, therefore, pick `null` for $\texttt{m}_0$ and `0` for $\texttt{c}_0$. The generated input executes the concrete path <1,2,3,4,8,9,10,12,15>, thus achieving the goal of executing statement 12.

In this example, SPA abstracts away all paths within `hasKey` and `getVal` and still provides simple and sufficient conditions for covering the goal—given an input generator as described. However, in general, the abstraction of terms in a path-family condition will produce an under-constrained (i.e., overapproximate) system of necessary but not sufficient conditions to symbolically describe a PFC or a variable. For example, an input generator might not be able to simultaneously satisfy all possible values of a PFC with the same concrete input, and thus it can miss the goal. (An example is the PFC for <11F> in Figure 1, for which no input guarantees that $*\texttt{.length} < \texttt{((m}_0\texttt{ =null)?0:m}_0\texttt{.size) + c}_0$.) One way to address this problem is to make the generator run multiple times and provide different solutions, increasing the chances that one of these inputs will cover the goal. An alternative is to make SPA refine the conditions (e.g., enter `hasKey`) and compute an (over)approximation of the effects of code previously abstracted away, thus reducing the imprecision of the under-constrained PFC and improving the chances that a generator will find an input that reaches the goal.

This example also shows that SPA, in contrast with traditional symbolic execution, traverses paths backwards and on demand when searching for definitions. For this reason, even if without abstractions, SPA might end up visiting far fewer paths than a "blind" forward symbolic execution. In the example, at the return statement 15, computing the precise symbolic value of `sz` does not require any auxiliary function except `hasKey`. Similarly, `m` does not require the subpaths in statements 11–14 (including the call to `append`) and `l` does not require `insert`. One consequence is that, for example, SPA does not have to traverse all combinations of paths in `append` and `insert`. In other words, instead of traversing a number of paths that is a product of the number of paths in both functions, SPA only traverses (at most) a sum of those paths.

3.2 Control-Dependence Families

SPA analyzes path families by grouping paths that share control dependencies into *control-dependence families*, or *cd-families*, that use the edges of the interprocedural control-dependence graph (ICDG) [34]. CD-families are more amenable than control-flow paths for multi-path analysis and abstraction for two reasons: a cd-family represents a (possibly infinite) group of paths, which can be partitioned on demand, and the ICDG edges in a cd-family have a one-to-one correspondence with the terms of the PFC.

To introduce cd-families, consider the example in Figure 1.[5] There is a potentially infinite number of paths from EN to node 12 because of loops that might exist in `hasKey`, `getVal`, or `insert`. Even if we ignore all function calls, there are still six paths to node 12 within `addElem`, distinguished by combinations of truth values at nodes 1, 4, and 6. However, no matter how these conditions in `addElem` evaluate, and no matter what occurs inside the auxiliary functions, all paths from EN reach node 11, and then only the CDG edge 11T is required to reach node 12. Therefore, the edge list <11T> precisely and concisely describes the PFC from EN to 12.

Another example is the cd-family between nodes 2 and 7 in the CDG of Figure 1, which represents the family of paths between nodes 2 and 7 in the program. Starting at node 2, there is no forward path in the CDG to node 7. However, after node 2 executes, the program returns to region START-T. Within this region, the program will then reach node 4 from which an edge list <4T,6T> of edges leads to node 7. Therefore, the PFC for the cd-family between nodes 2 and 7 is fully described by edge list <4T,6T>. In

[3] Replacing these return values with $*$ is an abstract interpretation [14]—a safe overapproximation—of the values that those functions can return.

[4] Because $*$ includes the `null` list, the first value of the PFC can produce an error, which we interpret as "not satisfied"—although, in this case, this is a spurious error.

[5] Auxiliary functions terminate and do not halt or throw exceptions.

other words, to find the cd-family, we started at node 2 and traversed the CDG backwards (through edge 1T) until we found a region in which, after returning from node 2, we found node 4, which is an ancestor of node 7. Edge 1T is not part of the definition of this PFC because it is traversed before reaching the starting node 2.

In general, the edges in a list for a PFC are not necessarily consecutive in the ICDG: the SPA technique often must partition a cd-family into constituent cd-families, each covering intermediate points (definitions). Such constituent cd-families represent subsets of the paths from the original cd-family. For example, in Figure 1, consider the paths between EN and 12 that also include node 7. Those paths form a cd-family whose PFC is described by edge list <4T,6T,11T>, where 11T does not follow 6T in the CDG.

To formally define a cd-family, we need two other definitions.

DEFINITION 1. Node u is a *control-ancestor* of node v if (1) there is a control-dependence region R in the ICDG that contains both v and u_a, where u_a is either u or an ancestor of u in the ICDG, and (2) v executes after u_a in region R.

For example, node 7 in Figure 1 is a control-ancestor of node 11 because the ancestor 4 of 7 in the CDG is in the same region (START-T) as 11, and node 4 executes before node 11.

DEFINITION 2. An edge list of control-dependence edges is *valid* for nodes s and t if (1) for each pair (e_1, e_2) of consecutive edges in that list, the first node u in the cd-region controlled by e_1 is the source node v of e_2, or u is a control-ancestor of v, (2) s is either the source node of the first edge in the list or a control-ancestor of that node, and (3) the first node in the cd-region controlled by the last edge is either t or a control-ancestor of t. For an empty list to be valid, s must be t or a control-ancestor of t.

We now define cd-family.

DEFINITION 3. A *cd-family* is a triple <s, t, E>, where s is the starting point, t is the ending point, and E is a collection of one or more valid lists of ICDG edges for the node pair (s, t).

To illustrate, the cd-family in Figure 1 of all paths from EN to 12 that include 7 is <EN,12,{<4T,6T,11T>}>. In this example, E contains one control-dependence edge list, but, in general, multiple edge lists might be needed to describe the desired paths between s and t. If, for example, an ICDG contains cycles, then E might contain an infinite number of edge lists. In such a case, we use regular expressions or, for the interprocedural case, context-free grammar expressions to produce a finite description of E.

3.3 The SPA Algorithm

In this section, we present the algorithm COMPUTESPA listed on Figure 2, which performs symbolic program approximation (SPA) of all paths between two program points, taking advantage of the control and data dependencies and using abstractions.

COMPUTESPA inputs a program P, a starting point s in P, an ending point t in P, a (possibly empty) set V of variables to symbolically evaluate at t, and an abstraction function $Abstract$. COMPUTESPA outputs the PFC to reach t from s and the symbolic values at t of the variables in V. Function *Abstract* decides when a variable in a given path is abstracted away as $*$. SPA treats the starting point s as the "entry" point for symbolic execution. Therefore, any variable x at the entry of s is treated as a symbolic input, becoming symbol x_0 at s.

SPA uses the interprocedural data dependencies in P in addition to the control dependencies required for cd-families. At line 1, SPA initializes *PFC(s→t)* with the description of the cd-family between s and t. This cd-family provides the top-level terms for *PFC(s→t)*; these terms will be later expanded into full symbolic expressions.

Algorithm COMPUTESPA

```
Input:  P: program to analyze
        s, t: start and end statements in P
        V: set of variables to evaluate at t
        Abstract: boolean function on use-path pairs
Output: PFC(s→t): path-family condition from s to t
        V_sym: symbolic values of variables in V

(1)       PFC(s→t) = computeCDFamily(s, t)
(2)       V_sym = {<u, PFC(s→t)> | u ∈ V × {t}}

(3)       workset = V_sym
(4)       foreach term C ∈ PFC(s→t)
(5)           p = getPrefixToCondition(PFC(s→t), C)
(6)           foreach use u ∈ C
(7)               workset ∪= <u, p>   // initially not expanded
(8)           endfor
(9)       endfor

(10)      while workset ≠ ∅
(11)          pick and remove <u, p> from workset

(12)          if Abstract(<u, p>)
(13)              linkToUse(<u, p>, <*, p>)
(14)              mark <u, p> expanded
(15)              continue to 10
(16)          endif

(17)          D = getReachingDefinitions(u, p)
(18)          I = getReachingInputs(u, p)
(19)          foreach assignment-path pair <a,p_a> ∈ D ∪ I
(20)              p' = getDefClearCoveringPath(p_a, var(u), p)
(21)              linkToUse(<u, p>, <a, p'>)
(22)              foreach use u_a ∈ rhs(a)
(23)                  if <u_a, p_a> not expanded: workset ∪ =<u_a, p_a>
(24)              endfor
(25)              foreach term C ∈ p'
(26)                  p_c = getPrefixToCondition(p', C)
(27)                  foreach use u_c ∈ C
(28)                      if <u_c, p_c> not exp. : workset ∪ =<u_c, p_c>
(29)                  endfor
(30)              endfor
(31)          endfor

(32)          mark <u, p> expanded
(33)      endwhile

(34)      return PFC(s→t), V_sym
```

Figure 2: The algorithm for computing a Symbolic Program Approximation (SPA) of all paths between two points.

For example, the initial cd-family between EN and 12 in the example of Figure 1 is <EN, 12, {<11T>}>, where 11T is the initial top-level term for *PFC(s→t)*. At line 2, SPA creates the set V_{sym} containing a pseudo-use of each variable in V at t (these pseudo-uses are necessary because not all variables in V might be used at t). Each pseudo-use is paired in V_{sym} with the condition required to reach t: *PFC(s→t)*. At line 3, SPA initializes the working set *workset* of use-path pairs (i.e., pairs of uses and the PFCs to reach those uses from s) with all use-path pairs from V_{sym}. In lines 4–9, SPA adds to *workset* all uses of variables occurring in the top-level terms (individual conditions) of *PFC(s→t)*, pairing each use u in those terms with the corresponding "prefix" PFC within *PFC(s→t)* that ends at the term where u is located, which is computed by *getPrefixToCondition*. For example, for a cd-family <s, t, {<aT,bT,cF>, <aF,dT>}> and term bT, *getPrefixToCondition* returns <s, t, {<aT,bT>}>, trimming the first edge list and discarding the second list because it does not contain bT.

After these initial steps, in lines 10–33, SPA proceeds to iteratively pick and remove from the workset one use-path pair <u,p> (line 11) and process that pair. The call to *Abstract* at line 12 decides whether this pair has to be abstracted away. If *Abstract* returns true, SPA does not process pair <u,p> normally; instead, in lines 13–15, SPA links to this use and PFC p the value $*$, marks the pair as "expanded", and continues to the next iteration in the *while* loop. In this case, no new elements are added to *workset*.

If, however, *Abstract* returns false, SPA proceeds to *backward-expand* u within p at lines 17–31. Backward expansion of a use u in path p finds all definitions and inputs (lines 17–18) for the variable used at u that might reach the location of u through some definition-clear path within the path family described by p. In lines 19–31, for each such definition or input a and its associated cd-family p_a (which ends at a), SPA computes cd-family p' (line 20) which describes all paths from s that reach a and then continue to u within the constraining p and without redefining the variable at u. The resulting pair of definition/input a and path p' is then linked to the working pair <*u*, *p*> (line 21).

Identifying the definitions and inputs for u and their reaching conditions is, however, not sufficient to obtain the symbolic value of u. All uses on the right-hand-side (i.e., defining) expressions of definitions, as well as all uses in the cd-family p', also have to be backward expanded transitively until only input and $*$ symbols are left. Therefore, in lines 22–24, SPA adds to *workset* every use u_a on the right-hand side[6] of a, associating each u_a with cd-family p_a, which ends at u_a. Similarly, lines 25–30 identify uses u_c in the terms of p', pairing each use with the prefix cd-family p_c that ends at that use, and adds those pairs to *workset*. Note that SPA only adds pairs to *workset* if they have not been already backward-expanded. Each use-path pair is marked as expanded at line 32.

When the *while* loop ends, all uses in *PFC(s→t)* and V, as well as all uses transitively reachable through definitions and sub-cd-families from those initial uses, have been expanded within their respective constraining cd-families or abstracted away. Implicitly, the linking process produces symbolic expressions for all variables of interest in terms of symbolic inputs and $*$. To obtain an explicit symbolic expression for *PFC(s→t)* and the variables in V, we traverse all links from use-path pairs to definitions and inputs, and continue through the use-path pairs on the right-hand-side expressions of reaching definitions and the PFCs for those definitions.

Safety and precision. The results of SPA are a safe overapproximation of PFCs and values of variables because, whenever the backward expansion of a use in COMPUTESPA is avoided, that use is assigned the overapproximating value $*$ instead. Rather than a single symbolic expression, SPA produces, in general, a multi-valued set of expressions; some values are guarded by *case* conditions, and the rest are not guarded (see Section 3.1). Therefore, when replacing symbolic inputs with concrete inputs, the resulting expression might evaluate to more than one concrete value.

Abstraction strategy. The use of the user-specified function *Abstract* in SPA makes this algorithm a framework that can be instantiated by any abstraction strategy represented by *Abstract*. An abstraction strategy might also access, in addition to the current use-path pair, any other information in program P or the status of COMPUTESPA at that point. (Moreover, an instantiation of SPA could also define how to pick the next <*u*,*p*> pair at line 11, for more control.) The abstraction strategy specified by the user must suit the particular analysis task. For example, the strategy might consist of abstracting certain library calls, as in the example of Section 3.1, or analyzing all code but stopping after SPA has transitively expanded a certain number of uses. In fact, the general strategy on which we will focus in the rest of this paper is to abstract path-condition sub-terms as $*$ which, if expanded, would be located below a limit e in the expression tree from the original PFC or the variables in V.

Termination. An important concern is the termination of the SPA algorithm. An abstraction might fail to prevent certain cyclic chains of uses (as found in loops) from expanding forever. Thus, a basic requirement for an abstraction strategy is to guarantee the termination of SPA. This is achieved, for example, by abstracting the values of uses so that they converge within loops, or by limiting the number of iterations in which a use is expanded. (The latter is the standard approach in literature for symbolic execution of individual paths [17]).

[6]The right-hand side for an input is empty.

4. APPLICATIONS

In this section, we discuss three areas of application of the symbolic approximation of programs with SPA: (1) analysis of changes for test-suite augmentation and test-case generation (Section 4.1), (2) computation of invariants and verification of properties (Section 4.2), and (3) approximation of the effects of module calls for incorporation into client analyses (Section 4.3).

4.1 Software Testing

In the context of regression testing, to test the effects of changes and augment a test-suite, it is necessary to analyze first the effects of changes and how these effects can propagate to the output. Current research uses *partial* symbolic execution to identify propagation conditions up to a certain distance from each change [2, 31]. The goal of this research is to analyze the longest possible distances within available computational resources; the longer the distance, the more likely it is that the effects of changes will continue to propagate after that distance and reach the output.

Unfortunately, the distances achieved experimentally by the resulting techniques are small, and the effects of changes do not always reach the output. The problem is that the effects of changes can stop propagating at places beyond the distance limit, which are missed by the analysis. Using SPA, however, we can "redistribute" the analysis effort: instead of analyzing the vicinity of a change with full precision, we can use SPA to analyze that same area with reduced precision and redirect freed computational resources to analyze areas beyond the typical distance limit that previous research can reach. Extending the reach of the analysis may cover many propagation-stopping points previously missed. In Section 5, we present an empirical study of this application of SPA.

SPA can also be used to generate inputs to target a specific goal, such as a change or a branch that remains untested. We showed such an application in Section 3.1, in which SPA computed an approximate PFC to cover statement 12 (branch 11T) in the example from Figure 1. The main difference with traditional test-case generation approaches based on symbolic execution is that SPA analyzes the conditions for all paths between the entry of the program and the goal, rather than individual paths. Path-family conditions can be simpler for families than for single paths, as long as a limited number variables need to be backward expanded—either because the coverage condition depends only on few intermediate variables, or because abstraction was applied. The limitation of the all-paths approach (i.e., SPA) is, of course, that abstractions might make the result too imprecise to achieve coverage of the goal. Fortunately, techniques based on abstractions, such as SPA, can be made to iteratively refine results in an effective way if good strategies can be designed to guide such refinements [5,9,25]. In all, we consider the all-paths approach to test-input generation as complementary to traditional symbolic execution and other input generation approaches; we plan to investigate this further in the future.

4.2 Invariants and Verification

There is a significant body of research on discovering program specifications. In particular, researchers have investigated static invariant discovery [22, 35] and dynamic invariant discovery [18]. These two methods use pre-defined invariant templates that are verified statically (i.e., for all possible executions) or checked dynamically (for a given test suite), respectively. More recent work has used symbolic execution to dynamically discover invariants directly from the program code [15]. SPA, in contrast, can be used to statically discover invariants from the program code (i.e., without restricting to templates) that are guaranteed to hold for all executions. For example, consider function `addElem` in Figure 1. During the computation of the path-family condition described in Section 3.1, we showed how SPA computes, in particular, an approximation of $\texttt{sz}_{11}$—the value of variable `sz` at Statement 11. This happens to be the same value that `sz` has at the return statement, $\texttt{sz}_{15}$, which is:

```
{ case m0 =null:  { 0, 1 }
  case m0 ≠null:  { m0.size, m0.size + 1 } }
```

This expression is an invariant that overapproximates the return value of function `addElem`. Manual examination of this function and its auxiliary functions reveals two imprecisions in this invariant: for $\texttt{m}_0$ =null, $\texttt{sz}_{15}$ cannot equal 0 because `hasKey(m,a)` will not return true for an empty map (note that SPA avoided the analysis of `hasKey`) and, for $\texttt{m}_0$ ≠null, the invariant does not ascertain whether the value is the original size of the map, or that size plus one. (Whether $\texttt{sz}_{15}$ is the first or the second value depends, again, on `hasKey`, which was abstracted away by SPA). Despite those two imprecisions, however, this invariant can be useful. If, for example, a developer specifies that `addElem` does not remove elements from the map and does not insert more than one entry, then the computed invariant is sufficient to verify that specification.

4.3 Modular Analysis

A major difficulty in static analysis of software arises from the potentially large size of the subject program or the difficulty of analyzing operations such as multiplications or calls to functions such as native methods in Java. The size of the program compromises the scalability of an analysis, especially for path-sensitive analyses, which suffer greatly from the path explosion problem. To address the scalability problem at least partially, researchers have looked at compositional analysis for symbolic execution by computing summaries of functions or code fragments before analyzing the code that invokes those functions or fragments [1,23]. Although symbolic summaries can help avoid multiple traversals of invoked modules, the complexity or impossibility of symbolically executing such modules with precision remains. Also, the complexity of the resulting composed expressions can become unmanageable.

SPA can be used to improve composability in symbolic execution by approximating the effects of modules, computing simplified summaries that could not be normally obtained with full symbolic execution. For example, consider again function `addElem` in Figure 1. The symbolic expression obtained for the return value is considerably simpler than the fully precise expression that would be obtained if we incorporated the effects of the call to `hasKey` in the path-family conditions. Furthermore, the effects of `addElem` on the map parameter `m` could also be approximated if SPA computes approximate summaries for sub-modules `hasKey` and `insert`.

5. EMPIRICAL EVALUATION

The goal of our study was to determine whether SPA can be more effective and more efficient than traditional symbolic execution. For that purpose, we implemented SPA and used it for one particular application: change analysis for test-suite augmentation.

In this section, we present our toolset and the results of this study. First, we describe our implementation of SPA and an tool based on SPA for test-suite augmentation (Section 5.1). Then, we describe our empirical setup (Section 5.2), the results of our study (Section 5.3), and threats to the validity of our study (Section 5.4).

5.1 Implementation

We implemented our SPA algorithm, given in Figure 2, as a tool of the same name: SPA. We implemented SPA in Java using the Soot Analysis Framework[7] to analyze programs in Java bytecode format. We also used DUA-FORENSICS, a tool that we developed previously, to perform the necessary data- and control-dependence analysis. The inputs for SPA are a program P (a set of Java class files), the starting point s and the ending point t of the analysis, and an abstraction strategy. To analyze realistic Java programs that use libraries and contain loops, SPA uses two mechanisms: (1) manually-encoded models of the effects of library calls to prevent the analysis from entering library code, and (2) user-provided limits for the maximum length of edge lists (see Section 3.2), the maximum size of the set of edge lists per path family, and the maximum number of iterations per loop. By default, SPA uses 10, 64, and 2 for these limits, respectively.

If we compare these limits with those used in traditional symbolic execution, we note that each control-dependence edge in a path family describes multiple statements (i.e., the statements in a control region). Such a limit can cover many more statements than, for example, the MATRIX-RELOADED tool used in Reference [31], which was normally limited to paths of 25 or fewer statements. Also, we note that using these limits make SPA incomplete by analyzing only a subset of all actual paths; we are currently working on a complete version of SPA that abstracts loops safely and deals with large numbers of edge lists.

Using SPA, we also implemented a tool called T_{SPA} for test-suite augmentation based on the technique from Reference [31]. The goal of this tool and technique is to compute testing requirements for a change corresponding to the conditions for propagation of the effects of that change to the output (i.e., the conditions for making the change observable). T_{SPA} inputs P and P', the original and modified versions of a program, respectively, and a distance limit d. Using DUA-FORENSICS, T_{SPA} identifies all dependence chains[8] of length at most d from the change. Then, for each dependence chain, T_{SPA} calls SPA separately for P and P', specifying in each case the starting and ending points of the chain as parameters s and t, respectively. In addition, T_{SPA} provides to SPA the specific path family between s and t to analyze, which is the subset of paths between s and t that cover the dependence chain.[9] For each of P and P', SPA returns the path-family condition (PFC) for covering each chain in that program and the symbolic value of the state variables modified along each chain. T_{SPA} then combines these results to obtain the conditions under which the effects of the change propagate at least up to distance d.

The abstraction strategy implemented by T_{SPA} that instantiates SPA is defined by a parameter e, corresponding to the maximum depth (Section 3.3), from the initial set of use-path pairs, to which these pairs are transitively expanded. In other words, e defines the maximum height of the resulting symbolic expression trees; all

[7] http://www.sable.mcgill.ca/soot/

[8] A *dependence chain* is a sequence of control and data dependencies where the first dependence starts at the change and the source statement of each next dependence is dependent on the previous dependence in the sequence. Dependence chains correspond to the sequence of events along which the effects of a change propagate. The union of the transitive closure of the dependence relation from a change corresponds to the forward slice from that change [37].

[9] Only a specific sequence of events between s and t covers a dependence chain.

variables below level e become $*$ (top). The value of e is inversely proportional to the level of abstraction.

5.2 Experiment Design and Setup

In this study, we applied three techniques to a set of changes in two different subjects and compared the effectiveness and efficiency of those techniques for test-suite augmentation. The techniques are:

1. MR, the technique defined in Reference [31] as implemented by the MATRIXRELOADED tool, which uses partial symbolic execution on each dependence chain. We use MR as a representative of traditional symbolic execution on control-flow paths, against which we compare SPA.
2. T_{SPA}-NoAbs, a version of T_{SPA} that assigns to the parameter e the value ∞ —no expansion limit and, thus, no abstractions. This version of T_{SPA} effectively implements the same technique from Reference [31], but using the SPA algorithm instead, without any abstraction.
3. T_{SPA}-Abs, which uses a value of e anywhere between 1 and 8, and thus, performs abstraction beyond this expansion limit.

To measure the effectiveness of these techniques on a change, we randomly generated a large number of test suites satisfying the testing requirements produced by that technique for that change and computed the percentage of those test suites that revealed a difference in the output; this percentage is our measure of effectiveness. To measure the efficiency of these techniques on a change, we recorded the time required by a technique to analyze a change and compute the testing requirements for that change; the longer a technique takes, the less efficient it is.

We chose our experimental subjects and changes from among those studied in Reference [31]. In that study, we experimented with six changes from Tcas, a small air traffic control program of 131 lines of code, and seven changes from two releases of NanoXML, an XML parser of 3497 and 4782 lines of code, respectively. To construct the test suites for each change, we used the pools of test cases provided with these subjects,[10] consisting of about 1600 and 215 test cases, respectively. For three changes in Tcas and five changes in NanoXML, the MR tool was not able to achieve 100% effectiveness within the available computational resources. In this paper, we study these eight changes to assess the improvements in effectiveness achieved by T_{SPA}-NoAbs and T_{SPA}-Abs for different values of the d parameter in both versions and the e parameter in T_{SPA}-Abs.

We performed our study of T_{SPA}-NoAbs and T_{SPA}-Abs on an Intel Core Duo 2 GHz machine with 2 GB of RAM. This configuration is the same used for the results previously obtained by MR, except that for this study, we had one extra GB of RAM, although in practice this was not required by T_{SPA}. For T_{SPA}-NoAbs and T_{SPA}-Abs, we set a time limit of two hours to each run on a change and combination of d and e. Whenever the analysis did not terminate within that time, we would stop it. We only report results for runs that finished within that time.

5.3 Results and Analysis

Effectiveness

Table 1 presents the test-suite augmentation effectiveness of each of the three implemented techniques (columns) on each of the studied changes (rows). For example, for change 3 in NanoXML, the effectiveness of MR, T_{SPA}-NoAbs, and T_{SPA} was 18.7%, 21.3%, and 26.1%, respectively. The effectiveness shown in Table 1 for each technique and change corresponds to the maximum effectiveness of all combinations of d and e for that technique and change.

Table 1: Difference-detection effectiveness for each technique.

change	MR	T_{SPA}-NoAbs	T_{SPA}-Abs
tcas 1	54.1%	100%	100%
tcas 2	18.6%	100%	100%
tcas 3	3.7%	100%	100%
nanoxml 1	67.2%	67.7%	92.8%
nanoxml 2	10.4%	39.9%	41.0%
nanoxml 3	18.7%	21.3%	26.1%
nanoxml 4	66%	100%	100%
nanoxml 5	42.2%	80.5%	81.9%

Table 2: Analysis time, in seconds, required to achieve maximum effectiveness.

change	T_{SPA}-NoAbs	T_{SPA}-Abs
tcas 1	7	7
tcas 2	6	6
tcas 3	6	6
nanoxml 1	3493	30
nanoxml 2	2939	28
nanoxml 3	3306	112
nanoxml 4	18	16
nanoxml 5	32	26

The first important result regarding effectiveness shown by this table is that, for all changes, T_{SPA} without abstractions (T_{SPA}-NoAbs) was more effective than MATRIXRELOADED (MR). In other words, the implementation of the technique from Reference [31] using the SPA algorithm without abstractions was at least as effective as the previous implementation of the same technique, MR, based on partial symbolic execution of individual control-flow paths. T_{SPA}-NoAbs was clearly more effective than MR on six changes: changes 1–3 in Tcas and 2, 4, and 6 in NanoXML. For change 5 in NanoXML, in particular, the effectiveness of the SPA-based version was 80.5%, almost twice as much as the 42.2% obtained by MR. Meanwhile, for change 3 in NanoXML, T_{SPA}-NoAbs was slightly more effective than MR (21.3% versus 18.7%), and, for change 3 in NanoXML, the difference was negligible.

We attribute the improvement in effectiveness of the implementation based on SPA without abstractions with respect to the MR implementation, at least in part, to the benefits of the SPA algorithm already described: analysis of path families instead of single paths and the use of data and control dependencies (i.e., slicing [37]) to partition these families only as required to cover a target—in this case, each dependence chain. Such benefits let the user set less stringent path-length and iteration limits in T_{SPA}-NoAbs than in MR, for similar computational budgets, so that more and longer paths can be analyzed.

The second important result from Table 1 is that, for all eight changes, the effectiveness of the T_{SPA}-Abs technique, using an abstraction strategy defined by a value of e between 1 and 8, was no less effective than the version without abstractions (T_{SPA}-NoAbs), despite performing a less precise, overapproximating symbolic execution of each dependence chain. In fact, T_{SPA}-Abs was remarkably more effective for change 1 in NanoXML and slightly more effective for changes 2, 3, and 5 of the same subject.

There are two reasons for the improvements in effectiveness of T_{SPA}-Abs with respect to T_{SPA}-NoAbs. First, T_{SPA}-Abs was able to analyze, for most changes, longer distances than T_{SPA}-NoAbs (i.e., distances for which T_{SPA}-NoAbs could not finish within the time limit). For example, for change 1 in NanoXML, T_{SPA}-NoAbs reached distance 5, whereas T_{SPA}-Abs could only reach distance 3[11]

[10]These subjects and test data are available at the SIR repository: http://sir.unl.edu.

[11]This difference is more remarkable than it seems. As dependence distances grow, the number of chains and paths can grow exponentially. For change 1 in NanoXML,

Second, for a given distance d, T_{SPA}-Abs was often able to finish within the time limit using greater values of the other practical limits discussed earlier (mainly, the maximum number of edge lists) than T_{SPA}-NoAbs was capable of handling. Thus, T_{SPA}-Abs was able to analyze, in many cases, more paths than T_{SPA}-NoAbs for the same distance. The information obtained by analyzing more and longer paths, although imprecise due to abstraction, was as effective or even more effective than the fully precise analysis.

Efficiency

In Table 2, we present the times in seconds that T_{SPA}-NoAbs and T_{SPA}-Abs required to obtain the results from Table 1, corresponding to the most effective of all combinations of parameters for which our toolset could finish within the time limit. For Tcas, all of the analysis times were equally small, but, for all changes in NanoXML, T_{SPA}-Abs took less time than T_{SPA}-NoAbs to compute the corresponding testing requirements. In particular, for the changes in which the most expensive analysis was necessary (i.e., changes 1–3 in NanoXML), the time required when using abstractions was remarkably lower than the time consumed by the precise version. For example, for change 1 in NanoXML, T_{SPA}-Abs required less than a hundredth of the time used by T_{SPA}-NoAbs—moreover, T_{SPA}-Abs was also more effective than T_{SPA}-NoAbs for this change.

For test-suite augmentation, a technique such as T_{SPA}-Abs that obtains similar or better results than the precise version with fewer resources can, in addition to saving such resources, be scaled to analyze more and longer dependence chains and paths covering those chains. If the produced requirements are not satisfactory for the developer, the saved resources may allow her to refine the current abstraction or try a different abstraction strategy.

Conclusions

Based on these results for test-suite augmentation, we conclude that, for these subjects and changes:

1. The implementation of the precise version of the technique from Reference [31] using SPA is more effective than the implementation based on traditional symbolic execution presented in that same reference.
2. SPA instantiated with abstractions is at least as effective, if not more effective, than the fully precise analysis (also implemented as a SPA instantiation).
3. SPA instantiated with abstractions can be much more efficient than the fully precise version of the same analysis, for a similar or greater effectiveness.

5.4 Threats to Validity

There is one main internal threat to the validity of the results obtained with the SPA tool. Although we tested our tool on several subjects, used multiple runtime checks, and manually examined many of the results, it is still possible that potential errors in the implementation of SPA affected in one way or another our results. A related threat corresponds to potential errors in our manual modeling of Java library methods.

The comparison with a different implementation of the test-suite augmentation technique (MATRIXRELOADED) raises an additional threat to the comparison of SPA with traditional symbolic execution: neither MATRIXRELOADED nor SPA are in a mature enough state of development to guarantee that, after the appropriate optimizations and bug fixes, the differences will remain as shown in this study. A comparison with other traditional symbolic execution tools might be necessary.

The main external threat is the limit in the size of our study. More subjects and changes have to be studied to generalize the conclusions summarized above. We are currently working on making SPA analyze new subjects and making it easier to set up experiments on additional sets of subjects and changes.

at distance 3, there are 65 chains; at distance 5, there are 1449 chains.

6. RELATED WORK

Path-sensitive control-flow analysis

Symbolic execution was introduced in the 1970's as a forward analysis of program paths [11, 28]. Clarke and Richardson later presented backward substitution as an alternative to forward expansion to perform symbolic execution, and called *global symbolic evaluation* the symbolic execution of all program paths [12]. In this paper, we presented a technique that effectively performs global symbolic evaluation of all paths between two points in a program and uses backward substitution. Our technique, however, operates on the dependence graph of the program instead of the traditional control-flow graph. Our technique also performs abstractions to reduce the cost of symbolic evaluation and make it more scalable.

Another path-sensitive static analysis approach is model checking [10]. Model checking explores all paths in state-transition systems to check or disprove properties. Recently, explicit-state model checking has been applied to software (e.g., [36]). In particular, software model checkers have been used to implement forward symbolic execution [27] by leveraging their path-exploration and backtracking capabilities. Our SPA algorithm analyzes multiple program paths too, but does so simultaneously and operates backwards on the dependence graph of the program to avoid traversing unnecessary combinations of subpaths and to enable abstraction.

Abstractions

Model-checking research has also used the idea of safely abstracting concrete, complex systems to facilitate verification on simpler, more manageable abstractions (e.g., [4, 7, 30]). However, when the checking process fails, researchers use infeasible counterexamples to refine the current abstraction (e.g., [3–5, 7, 9, 25]). Model-checking techniques typically start with the interprocedural control-flow graph of the program as the initial abstraction[12] and, by default, do not perform symbolic evaluation other than tracking sets of predicates and symbolically executing counterxamples to check their feasibility. Like these techniques, SPA uses overapproximating abstractions. However, SPA works on families of control-dependence paths, which naturally represent the terms of symbolic path conditions between two points. SPA partitions these path families guided by data dependencies instead of counterexamples. Furthermore, SPA separately partitions path families that correspond to different terms, which avoids combining subpaths unnecessarily, as shown in Section 3.1. These features of SPA reduce the number of paths analyzed with respect to techniques based on control-flow.

Path slicing [26] has been proposed to remove irrelevant portions of a counterexample path and perform only relevant abstraction-refinements for model checking with respect to that path. Interestingly, the sliced path implicitly represents an individual family of all paths that contain the preserved portions of the original path. While the resulting path family corresponds to a control-dependence path family as used in SPA, there are, in general, multiple path families between two program points, all of which are analyzed by SPA. Moreover, SPA iteratively partitions such path families into sub-families to obtain the desired level of abstraction, whereas path slicing simply feeds the sliced path back to the model checker for refinement, without maintaining any symbolic representation of the program.

Uninterpreted functions represent operations or code segments

[12]Calysto [3] improves performance by starting with the call graph instead and abstracts function calls, which it later expands with their control-flow paths as needed.

for which nothing is known. They have been used together with specialized logics (e.g., [8]) to abstract classes of constructs in system designs for verification. Researchers have also used uninterpreted functions in symbolic execution when comparing two programs to prevent the unnecessary analysis of common segments [16, 29, 33]. The value $*$ in SPA can be considered an uninterpreted function with a particular meaning: top. This is the same meaning assigned by some analyses (e.g., [7]) to unsupported operations such as multiplication. SPA, however, uses $*$ more generally to abstract away arbitrary subterms within symbolic expressions.

Fischer et al. [21] presented a hybrid analysis in which a standard data-flow analysis is iteratively enriched with predicates, suggested by counterexamples, that add path sensitivity. SPA, in contrast, starts with a control-dependence based path-sensitive analysis and later abstracts subterms arbitrarily (e.g., based on resource bounds) to provide an overapproximate path-sensitive result. SPA, in that sense, can be seen as a specific form of analysis with dynamic precision adjustment [6], but working on control dependencies instead of control flow.

Taghdiri et al. [35] presented an algorithm that mixes data-flow analysis with symbolic execution to compute static invariants. SPA can also be used to compute invariants, but provides a finer-grained control over the abstraction level and eliminates unnecessary combinations of subpaths that arise from control-flow analysis.

Test-input generation

As mentioned earlier, model-checking techniques can be used for test-input generation—an application we discussed for SPA. More specialized test-input generation techniques based on symbolic execution include "dynamic" symbolic execution [24, 32, 38]. We believe that our all-paths approach complements these approaches, which are based on the smart exploration of individual paths. We also expect that modular analysis using SPA will improve compositional specializations of those techniques [1, 23].

7. CONCLUSION AND FUTURE WORK

In this paper, we presented Symbolic Program Approximation (SPA), a technique that performs symbolic execution of multiple paths between two points, using a combination of dependence analysis and backward expansion on those dependencies. Our technique works directly on program dependencies to analyze only the parts of the program that are needed, thus avoiding unnecessary intermediate (sub)paths. In addition, our technique uses abstraction to reduce the cost of the backward expansion process, trading precision for scalability. The result of SPA is a safe overapproximation of the set of program paths between two points.

We also presented a tool of the same name—SPA—that implements this framework and supports the instantiation of precise and approximate analyses of changes for test-suite augmentation. Using this tool, we experimented on a set of program changes and concluded that SPA can serve as a better implementation for the precise analysis of changes, while SPA can also be applied with abstractions to obtain results as effective or more effective in less time. This experiment supports our hypothesis that applications of symbolic execution on all paths can benefit from using abstractions.

In the future, we plan to empirically investigate the use of SPA in test-input generation, modular analysis, and invariant detection. Also, because SPA currently performs a very simple abstraction by replacing variables with the $*$ value (i.e., top), we intend to leverage abstractions of intermediate complexity such as those from shape analysis [30], predicate abstraction [4], and the Synergy and Dash approaches [5, 25].

Acknowledgements

This work was supported in part by NSF awards CCR-0306372, SBE-0123532, and CCF-0725202 to Georgia Tech and by an award from Tata Consultancy Services.

8. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Mar. 2008.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. of Testing and Academic Industrial Conf. Practice and Research Techniques*, pages 137–146, Aug. 2006.

[3] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 211–220, May 2008.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 2001.

[5] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. of the Int'l Symp. on Softw. Testing and Analysis*, pages 3–14, July 2008.

[6] D. Beyer, T. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. Sept. 2008.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[8] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV 02*, pages 78–92, July 2002.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[11] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Softw. Eng.*, 2(3):215–222, Sept. 1976.

[12] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, February 1985.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, . Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, May 2000.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Symp. on Principles of Prog. Lang.*, pages 238–252, Jan. 1977.

[15] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proc. of the Int'l Conf. on Softw. Eng.*, May 2008.

[16] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic

execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, 2006.

[17] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. of Int'l Conf. on Automated Software Eng.*, pages 157–166, 2006.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Softw. Eng.*, 27(2):99–123, Feb. 2001.

[19] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[20] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, July 1987.

[21] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, Sept. 2005.

[22] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. of the Int'l Symp. of Formal Methods Europe on Formal Methods for Increasing Softw. Productivity*, pages 500–517, Mar. 2001.

[23] P. Godefroid. Compositional dynamic test generation. In *Proc. of the Symp. on Principles of Prog. Lang.*, pages 47–54, Jan. 2007.

[24] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 2005.

[25] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. of the Int'l Symp. on Foundations of Softw. Eng.*, pages 117–127, 2006.

[26] R. Jhala and R. Majumdar. Path slicing. In *PLDI '05*, pages 38–47, June 2005.

[27] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of Ninth Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, April 2003.

[28] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. of the Int'l Symp. on Foundations of Softw. Eng.*, pages 226–237, Nov. 2008.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[31] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. of Int'l Conf. on Automated Softw. Eng.*, pages 218–227, Sept. 2008.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of 5th Joint Meeting of the European Softw. Eng. Conf. and Int'l Symp. on the Foundations of Softw. Eng.*, pages 263–272, Sept. 2005.

[33] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, Apr. 2008.

[34] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10(2):209–254, 2001.

[35] M. Taghdiri, R. Seater, and D. Jackson. Lightweight extraction of syntactic specifications. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 276–286, 2006.

[36] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Eng.*, 10(2):203–232, 2003.

[37] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.

[38] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, June-July 2009.