Staged Symbolic Execution

Junaid Haroon Siddiqui
University of Texas at Austin
Austin, TX 78712
jsiddiqui@utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712
khurshid@ece.utexas.edu

ABSTRACT

Recent advances in constraint solving technology and raw computation power have led to a substantial increase in the effectiveness of techniques based on symbolic execution for systematic bug finding. However, scaling symbolic execution remains a challenging problem.

We present a novel approach to increase the efficiency of symbolic execution for systematic testing of object-oriented programs. Our insight is that we can apply symbolic execution in *stages*, rather than the traditional approach of applying it all at once, to compute *abstract symbolic inputs* that can later be shared across different methods to test them systematically. For example, a class invariant can provide the basis of generating abstract symbolic tests that are then used to symbolically execute several methods that require their inputs to satisfy the invariant. We present an experimental evaluation to compare our approach against KLEE, a state-of-the-art implementation of symbolic execution. Results show that our approach enables significant savings in the cost of systematic testing using symbolic execution.

1. INTRODUCTION

Symbolic execution is a technique for systematic exploration of program behaviors using symbolic inputs, which was first developed over three decades ago [8, 18]. While several researchers used symbolic execution to enable different kinds of analyses since its inception, it is only during the last decade that techniques based on symbolic execution have started to realize the promise its powerful analysis holds for systematic bug finding [6, 7, 14, 15, 17, 21]. A key driving factor behind this resurgence of symbolic execution is advances in constraint solving technology and substantial increase in raw computation power, which are needed to solve *path conditions* – constraints on inputs that execute a particular program path – a fundamental element of symbolic execution. While these technological advances have fueled the development of novel techniques based on symbolic execution, scaling it remains a challenging problem.

In this paper, we present a novel technique for efficient symbolic execution of object-oriented programs with contracts, such as class invariants and pre-conditions, to facilitate systematic testing. Specifically, we consider the problem of how to use symbolic execution to systematically enumerate inputs for a Java method m that has a pre-condition p, which is also written in Java as a *repOk* predicate, i.e., a method that inspects its inputs to check the pre-condition and returns true if and only if it is satisfied. Our goal is to provide efficient and effective enumeration of *valid* inputs for m, i.e., inputs that satisfy the pre-condition. The key technical challenge in solving this problem is to lead symbolic execution into the body of m, which must be preceded by an invocation of *repOk*, since correct behavior of m requires the pre-condition to hold. Thus, symbolic execution must be performed on the program "`if (repOk()) m();`".

This problem of test generation using pre-conditions has been studied in various research projects over the last decade [3, 11, 17, 19, 21] using two primary approaches: black-box testing and white-box testing. In black-box testing [3,11,19], the pre-condition is taken in isolation of the method under test and used to enumerate valid concrete inputs, which are later used to test the method. The key advantage of this approach is its ability to generate *dense suites*, which allow bounded exhaustive testing, which has been shown to be effective at finding bugs in a variety of applications, including compilers [12], refactoring engines [9], service location architectures [19], and fault-tree analyzers [23]. A basic limitation of this approach however is the need to generate a large number of tests and the need to run each of those tests against all methods under test — even if for certain methods, many tests are equivalent.

In contrast, in white-box testing, the program "`if (repOk()) m();`" is directly used for test generation, e.g., using symbolic execution [17, 21]. The key advantage of this approach is its ability to directly explore a large number of paths in the method under test and to generate test suites that achieve high code coverage and likely contain fewer tests than black-box approaches. A basic limitation of this approach however is the need to repeatedly consider symbolic execution of the *repOk* method for each method under test – by construction, symbolic execution must execute "`if (repOk()) m();`" for each m that has *repOk* as its pre-condition. Thus, this approach requires enumeration of valid inputs from scratch for each method under test — even if certain methods have the same pre-condition. While re-use of concrete inputs generated for one method, say m, to test another method, say m', is possible, this white-box

SAC'12 March 26-30, 2012, Riva del Garda, Italy.

approach for method m then degenerates into a black-box approach for method m'.

Our insight is that we can apply symbolic execution in *stages*, rather than the traditional approach of applying it all at once, to compute *abstract symbolic inputs* that can later be shared across different methods to test them systematically. The first stage performs symbolic execution of *repOk* to generate abstract symbolic tests, which are object graphs that have symbolic components defined by constraints, akin to path conditions in symbolic execution. Thus, one abstract symbolic test represents (possibly) many concrete tests, and the suite of abstract symbolic tests compactly represents a likely much larger suite of concrete tests. The second stage takes, in turn, each method under test that has the same pre-condition (as defined by *repOk*), and symbolically executes it using each abstract symbolic test.

Symbolic execution during the second stage dynamically expands each abstract symbolic test into a number of concrete tests based on the control-flow of the method under test. Methods that require more tests due to complex control-flow are tested using more tests and methods that require fewer tests are tested against fewer tests. While this control-flow-driven exploration during the second stage allows our approach to share benefits of white-box techniques, the use of specifications in the first stage enables our approach to share a key benefit of black-box techniques: generation of abstract symbolic tests can proceed even before the code to test is implemented, much in the spirit of test-first programming — this contrasts with other approaches based on symbolic execution, which a priori require an implementation of code under test. We believe our approach provides a sweet-spot for using symbolic execution for systematic testing of methods with pre-conditions.

A major advantage of our approach is the re-use of abstract symbolic tests across different methods. The re-use further increases in the context of software evolution. Abstract symbolic tests are generated with respect to specifications. If some specifications remain unchanged while code evolves, the same abstract symbolic tests can be re-used for the new version of code. Thus, our approach holds potential for significant savings in regression testing effort.

Our work opens a novel avenue for tackling the problem of scaling symbolic execution. Our experimental evaluation demonstrates the potential of our approach in the context of well-studied subject programs. However, abstract symbolic tests are not limited to object graphs with constrained symbolic components. To illustrate, an abstract symbolic test may represent a partial XML document or a partial Java program with symbolic elements that have constraints, thereby enabling a novel approach for testing of systems that take XML or Java programs as inputs, such as compilers and refactoring engines. We believe staged symbolic execution and abstract symbolic tests provide precisely the technical elements that are needed to take approaches based on symbolic execution, such as dynamic symbolic execution [6, 14, 24], aka concolic execution [21], to a higher level of efficiency and effectiveness.

This paper makes the following contributions:

Abstract symbolic tests. We introduce abstract symbolic tests that are a combination of concrete tests and constrained symbolic elements. These tests are not executable using conventional execution, rather they provide a basis for symbolic execution of the program with pre-conditions.

```
class SearchTree {
  class Node {
    int data;
    Node* left;
    Node* right;
  };
  Node* root;
  int size;
 public:
  bool repOk() {
    // class invariant
  }
  void add(int data) {
    // method under test
  }
};
```

Figure 1: Skeleton code for a binary search tree.

Staged symbolic execution. We introduce the technique of performing symbolic execution in stages, where the first stage generates abstract symbolic tests, which the second stage uses for systematic testing — each abstract symbolic test is dynamically expanded into a number of concrete tests depending on the control-flow complexity of the program under test. Staged symbolic execution allows both a reduction in test suite size without a loss in its quality, as well as a novel re-use of tests.

Evaluation. Our prototype implementation uses KLEE [7] — an open-source tool for symbolic execution that has been used by a variety of users in academia and industry — as enabling technology. We present an experimental evaluation to compare our approach against the standard approach of KLEE using well-studied subjects that implement complex data structures. Results show that our approach holds significant benefits for increasing efficiency and effectiveness of systematic testing using symbolic execution.

2. MOTIVATIONAL EXAMPLE

Consider a binary search tree (Figure 1). Its structural properties are acyclicity along `left` and `right` pointers, a correct `size`, and proper ordering of data elements in the nodes (i.e. each node has smaller elements in the left subtree and larger elements in the right subtree). We will test the `add` method for up to three nodes in a *bounded-exhaustive* manner. Bounded exhaustive search tests the method for *all* valid inputs within the given bounds. As shown in Figure 2, there are nine *non-equivalent* binary search tree object graphs of at most three nodes.

To test the `add` method using a black box technique such as Korat [3], we need to provide the specification of bounds on input variables.

Black-box testing using Korat generates tests using these bounds and the `repOk` class invariant. With a bound specification of 3 nodes and 4 integers, there are more than four million permutations. Korat finds all instances that are satisfied by the class invariant using an efficient pruning based search. The number of valid binary search trees using four integers is 53. There are 9 unique object graphs and each of these 53 trees is an assignment of data values to one of these object graphs.

To exhaustively test the `add` method of binary search tree for three nodes using a black-box approach, we need to store 53 tests and run 53×4 = 212 tests (there are 4 distinct values that can be added to the tree). To test the same method

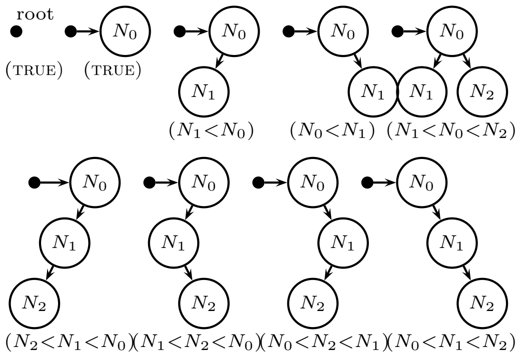

Figure 2: *Abstract symbolic inputs* consist of an object graph/path condition pair.

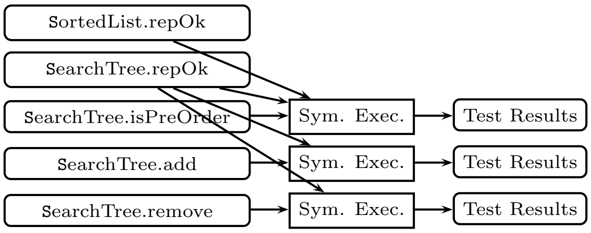

Figure 3: repOk symbolically executed as part of every method tested with symbolic execution.

using symbolic execution, we do not need to provide a bound on any field. For the add method, we write the following code and run it through symbolic execution.

```
SearchTree b;
if (b.repOk()) { // pre-conditions
  try {
    b.add (x); // x is a symbolic integer
    // check b using post-conditions
  } catch(...) {
    // report implementation bug
  }
}
```

Based on the symbolic execution interpreter used, we may also need to add code to explicitly mark the fields as symbolic. For example, for lazy initialization [17], we write getters functions for each reference field that lazily initialize a field from its field domain at first access.

The essence of this technique is that test generation and test execution are combined as one. Symbolic execution of repOk prunes invalid choices and continues to the method under test for valid choices. If the tests are concretized after repOk, symbolic execution will provide one valid instance for each object graph. However, it may not cover all paths in the method under test as symbolic execution was unaware of the path condition(s) that *will* be formed during the execution of the method under test.

This provides our motivation for a technique that can store abstract tests with a symbolic state. This symbolic state should be restored when the test is executed and non-reference fields can be concretized as needed during symbolic execution of the method under test. Such a technique would combine the benefits of both white box and black box testing for methods with structurally complex inputs.

3. STAGED SYMBOLIC EXECUTION

In this section, we discuss the high-level architecture of staged symbolic execution in contrast to standard symbolic execution and define basic concepts. We then discuss the techniques to create abstract symbolic inputs and to regenerate the symbolic execution state using them.

3.1 High-level architecture

The high-level architecture of staged symbolic execution is explained with three example methods. We consider the add, remove, and isPreOrder methods of a binary search tree. The first two add and remove an element respectively while the third takes a sorted linked list and checks if it contains the pre-order traversal of the binary search tree. In Figure 3, the high level steps involved in their symbolic execution are shown. The symbolic execution engine takes the class invariants for the concerned object(s) *and* the method under test and explores the *complete* search space. The class invariant is symbolically executed whenever a method needs a valid object of that type.

Staged symbolic execution differs in that symbolic execution of one class invariant is only done once. Figure 4 shows the high level steps involved. The invariants are symbolically executed in one stage while the second stage needs to read the explored valid results and simply proceed with symbolic execution of the method under test. This enables a re-use of the symbolic exploration of the class invariants.

3.2 Abstract symbolic inputs

We define an *abstract symbolic input* as a tuple $\langle o, p \rangle$ where o is a rooted object graph and p is a path condition over fields in the object graph o. Figure 2 shows eight abstract symbolic inputs for a binary search tree with a bound of three nodes. Each abstract symbolic input consists of a concrete object graph with a path condition on its fields. To generate these abstract symbolic inputs, we can use code like this:

```
SearchTree b;
if( !b.repOk())
  SEE_ignore();
```

SEE_ignore is a symbolic execution engine to backtrack the current search. When the above code is run using a symbolic execution engine it produces all valid object graphs along with their path conditions as shown in Figure 2.

An *abstract symbolic test driver* as a test function that utilizes abstract symbolic inputs and possibly other concrete and/or symbolic inputs to invoke a method under test. When the driver is symbolically executed, it uses abstract symbolic inputs to run *abstract symbolic tests.*

To test the add method of SearchTree, our running example, we can use the abstract symbolic test driver below.

```
void testSearchTreeAdd() {
  SearchTree* b=regenSymObjGraph<SearchTree>
                         (seedListOfSearchTree);
  try {
    // b already satisfies pre-conditions
    b->add (x); // x is a symbolic integer
    // check b using post-conditions
  } catch(...) {
    // report implementation bug
  }
}
```

3.3 Creating abstract symbolic inputs

To create abstract symbolic inputs, we need to save enough information to later reconstruct the symbolic execution state.

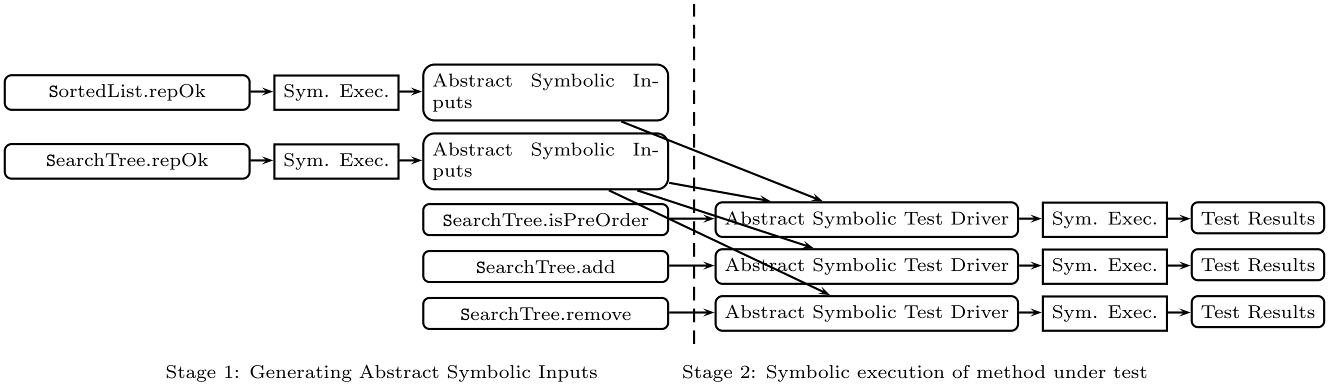

Figure 4: Staged symbolic execution of add, remove, and isPreOrder functions of binary search tree.

A naive solution is to store the path condition as is. If the underlying symbolic execution engine supports pointers, then the path condition contains equality constraints for pointers involved in the object graph. For other symbolic execution engines, a scheme like lazy initialization [17] can be used. In this case, the path condition has equality constraints for integers that choose the pointer from a pool of pointers.

We can improve the naive solution by serializing the object graph as it exists during generation and remembering mapping of fields to variables in the path condition. This would be akin to storing it as shown in Figure 2. However, this makes regeneration complex. After instantiating the object graph, regeneration would require creating the symbolic variables involved in it and adding the constraints over them.

The above solution is practical, but there is an even easier approach that requires minimal changes in a symbolic execution engine to support abstract symbolic inputs. This approach forgoes storing the path condition and object graph separately (as shown in Figure 2) and only stores one solution to each path condition (e.g. $root = N_0, N_0.right = N_1, N_0.data = 1, N_1.data = 2$ etc.). Although, we lose information in this scheme, we show in the following section that when combined with the original class invariant, it is easy to reconstruct abstract symbolic inputs. It makes generating abstract symbolic inputs much easier.

To create abstract symbolic inputs for a class C, the user only provides a deterministic side-effect free class invariant (repOk). The following code is automatically generated and symbolically executed.

```
void generateC() {
  C c;
  if(!c.repOk())
    SEE_ignore();
}
```

SEE_ignore is a symbolic execution engine function that backtracks from the current search. Each completed path is solved for a satisfying solution (if one exists) and stored as an abstract symbolic input. Note that this is not the same as using concrete tests. This concrete representation is generated by a symbolic execution engine and when given to the same symbolic execution engine it can be used to retrace the execution path and regenerate the symbolic state. This is discussed in the next section.

As an additional benefit of storing a satisfying solution instead of a path condition, we have enabled the possibility to use other tools like Alloy [16] or Korat [3] for generating tests, while still using symbolic execution to realize them as abstract symbolic inputs. Similarly, a symbolic execution engine that supports pointers (e.g. CUTE [21]) or one with lazy initialization (e.g. JPF [25]) can be used for generation and another one can be used for execution that might better support unmodified large programs (e.g. KLEE [7]). Even the programming languages used for generation and execution predicates can be different, if the class invariant is written in both languages.

3.4 Regenerating symbolic execution state

To regenerate symbolic execution state from abstract symbolic inputs stored as a satisfying assignments to path conditions, we introduce a new algorithm. Symbolic execution engines did not need to be modified for generating abstract symbolic inputs because we stored them as a satisfying assignment, and symbolic execution engines depend on the ability to solve path conditions. However for reconstructing abstract symbolic inputs, we need to modify the symbolic execution engine. A naive solution to regenerate is to take the stored values as concrete and run the method under test using them. However, this defeats the purpose as we cannot explore all possibilities for the method under test that were possible using the path conditions. Thus we need to rebuild the symbolic state.

The technique we use utilizes *seeds*. Seeds are given to a symbolic execution engine to start its search. A seed consists of a tuple of values which provide the initial value for new symbolic variables formed. Some symbolic execution engines [14,21] collect the path constraint for a complete execution using the seed values and then explore other branches by negating clauses in the path constraint. On the other hand, forward symbolic execution [7, 8, 18] checks on every branch that a satisfying seed exists to either prioritize that branch or exclude other branches altogether. We describe our technique for forward symbolic execution. It can be adapted for other tools as well.

Our algorithm in Figure 5 works in the context of a symbolic execution engine. When the code under test requests loading abstract symbolic inputs, the function regenSymObj (line 3) gets invoked. It takes the current symbolic state, a list of seeds where each seed is a solution to one path condition, and an object of the type to be generated. After this function returns, the fields of this object are constrained

```
1:  globalPruningEnabled ← FALSE

2:          ▷ Invoked by the user to load abstract symbolic inputs.
3:  function regenSymObj(state, seedList, symObject)
4:      assert(isEmpty(getSeedSet(state)))
5:      for all $f$ ← seedList do
6:          addToSet(getSeedSet(state), $f$)
7:      end for

8:      globalPruningEnabled ← TRUE
9:      executeSymbolic(symObject→repOk)
10:     globalPruningEnabled ← FALSE
11: end function

12: ▷ Invoked by symbolic execution engine whenever a new branch
    is seen. Branch is pruned if it returns false.
13: function newBranch(state)
14:     if globalPruningEnabled then
15:         for all $f$ ← getSeedSet(state) do
16:             if isSolvableUsingSeed(state, $f$) then
17:                 return TRUE
18:             end if
19:         end for
20:         return FALSE
21:     end if
22:     return TRUE
23: end function
```

Figure 5: Regenerating abstract symbolic inputs.

by satisfying the class invariant.

The *seed set* (set of solutions to path conditions – now to be used as seeds for new symbolic variables) in the current symbolic execution state should be empty (line 4). If it is not empty it means a previous abstract symbolic input was not instantiated correctly.

On lines 5-7, we load the actual seeds from a list of seeds into the seed set of the current execution state. After that we enable pruning of unnecessary branches (line 8) and reset it to enable exploring everything (line 10) after symbolically executing the class invariant of the symbolic object being constructed (line 7).

The NewBranch function (lines 13-23) is always called internally by the symbolic execution engine. If it returns false, the branch is pruned out. When pruning is enabled (line 14), it returns false (line 20) when there is no seed that satisfies the current execution state. New constraints are formed with each new branch, possibly resulting in some constraints becoming infeasible for every seed. Such branches are pruned out. These have been tested when the abstract symbolic input was generated and it turned out repOk returns false or they become infeasible. isSolvableUsingSeed function (line 16) checks if the current path condition is necessarily false using the values in the given seed.

The changes required to support regenerating abstract symbolic inputs in a forward symbolic execution engine are: (1) a set of seed values which is used by new symbolic variables; (2) a function to decide if new branches are to be explored; and (3) the algorithm in Figure 5 to load seed values and temporarily enable branch pruning.

To reconstruct abstract symbolic inputs, the *user* invokes regenSymObjGraph in an abstract symbolic test driver (see 3.2). This invokes the internal RegenSymObj function for the current symbolic execution state. KLEE [7] – the engine we used – supports seeds at the start of symbolic execution and allows restricting explored branches to these seeds. We modified it to enable adding seeds dynamically and restricting exhaustive branch exploration temporarily (for the duration of symbolically executing the class invariant).

4. EVALUATION

We implemented staged symbolic execution on top of the KLEE symbolic execution tool [7]. To evaluate our approach, we consider five methods from a binary search tree, six methods from a sorted linked list, and two methods of a binary heap. We compare staged symbolic execution to standard symbolic execution using KLEE and black box bounded exhaustive testing using Korat. The experiments were performed on a 2.53GHz dual core i5 machine with 8GB of memory. We show how abstract symbolic tests enable small but effective test suites for methods with structurally complex arguments. We also show that testing time for a set of methods of the same type, and for methods with more than one structurally complex argument is substantially reduced. The following subsections go over different experiments we performed in detail.

4.1 One structurally complex argument

We test the add and remove methods of a binary search tree, a sorted linked list, and a binary heap – all taking a single structurally complex arguments (see first six method entries of Table 1).

The "Stage 2 Savings" column shows the savings in time, in comparison to standard symbolic execution assuming that Stage 1 has been done for another method (e.g. we are testing *add* and we have Stage 1 results from testing *remove* or we are doing regression testing and we have Stage 1 results from a previous run). From the results we can see that staged symbolic execution saves us from exploring a number of states repeatedly and stage 1 results can be readily used for testing other methods.

4.2 Multiple independent arguments

The last five entries of Table 1 are methods with two and three structurally complex independent arguments. We consider isEqual and isPreOrder methods of binary search tree. The former considers two binary search trees, while the later takes a binary search tree and a sorted linked list. We also take isEqual, merge, and isIntersection methods from a sorted linked list. The first two take two sorted linked lists as arguments, while the last takes three sorted linked lists.

The added advantage of staged symbolic execution with multiple independent arguments is that Stage 1 can only be run using one argument, and Stage 2 can use the results repeatedly. Standard symbolic execution has to symbolically explore the class invariant of the second argument for every valid instance of the first argument.

Note that multiple dependent arguments are the same as single argument as they can be generated using a repOk that invokes the repOk of both arguments. We therefore do not evaluate them separately.

To show how staged symbolic execution scales with varying number of arguments, we consider a union method of a binary search tree and of a sorted linked list. We test taking the union of two, three, and four arguments. and present the results in Table 2.

4.3 Regression testing using mutants

In this experiment, we show the benefit of staged symbolic execution for regression testing. Following Rothermel et al. [10], we use mutants to simulate software evolution. We generated mutants manually with mutation operators: changing a comparison operator, changing a field

Table 1: **Comparison of standard symbolic execution and staged symbolic execution.**

				Staged Symbolic Execution		
benchmark	max.	Total	Symbolic Execution	Stage 1	Stage 2	
	size[1]	non-equiv. tests[2]	Valid/Explored tests (Time)	Valid/Explored tests (Time)	Valid/Explored tests (Time)	Stage 2 Savings
SearchTree.add	3	375	29/92 (34.2s)	9/72 (33.1s)	29/29 (9.0s)	3.8X
	4	5,955	99/344 (7m54.7s)	23/268 (6m26.5s)	99/99 (1m16.1s)	5.1X
	5	76,062	351/1298 (87m32.4s)	65/1012 (71m24.0s)	351/351 (12m29.2s)	5.7X
SearchTree.remove	3	375	49/112 (1m00.7s)	Reused	49/49 (21.7s)	2.8X
	4	5,955	175/420 (13m48.2s)	Reused	175/175 (3m42.1s)	3.7X
	5	76,062	637/1584 (151m52.2s)	Reused	637/637 (38m37.8s)	3.9X
SortedList.add	6	12,012	28/56 (51.5s)	7/35 (44.9s)	28/28 (8.6s)	6.0X
	7	51,480	36/72 (3m02.4s)	8/44 (2m58.2s)	36/36 (15.0s)	12.2X
	8	128,790	45/90 (13m09.3s)	9/54 (14m01.6s)	45/45 (33.1s)	23.8X
SortedList.remove	6	12,012	28/56 (1m18.9s)	Reused	28/28 (8.6s)	9.2X
	7	51,480	36/72 (4m23.4s)	Reused	36/36 (16.4s)	16.1X
	8	128,790	45/90 (16m35.6s)	Reused	45/45 (1m04.7s)	15.4X
BinaryHeap.add	8	6,937,713	26/135 (1m57.2s)	9/118 (1m41.7s)	26/26 (20.1s)	5.8X
	9	83,510,790	29/165 (3m06.6s)	10/146 (2m47.4s)	29/29 (27.6s)	6.8X
	10	988,213,787	32/198 (4m37.0s)	11/177 (4m28.2s)	32/32 (35.1s)	7.9X
BinaryHeap.remove	8	6,937,713	13/122 (2m18.2s)	Reused	13/13 (23.1s)	6.0X
	9	83,510,790	14/150 (3m14.9s)	Reused	14/14 (34.5s)	5.6X
	10	988,213,787	15/181 (4m49.4s)	Reused	15/15 (41.5s)	7.0X
SearchTree.isPreOrder	2,2	375	16/70 (28.8s)	14/92 (44.0s)	16/16 (5.7s)	5.1X
	3,3	27,636	48/310 (8m25.4s)	13/86 (39.2s)	48/48 (58.7s)	8.6X
	4,4	2,623,995	149/1389 (157m10.9s)	28/288 (6m37.4s)	149/149 (11m23.8s)	13.8X
SearchTree.isEqual	2,2	625	16/96 (53.2s)	Reused	16/16 (8.9s)	6.0X
	3,3	108,241	81/711 (46m37.5s)	Reused	81/81 (4m47.1s)	9.7X
	4,4	28,100,601	Timeout[3]	Reused	529/529 (293m10.02s)	N/A
SortedList.merge	3,3	7,056	69/119 (1m05.3s)	Reused	69/69 (15.8s)	4.1X
	4,4	245,025	251/341 (4m36.3s)	Reused	251/251 (1m26.1s)	3.2X
	5,5	9,018,009	923/1070 (24m32.9s)	Reused	923/923 (17m19.1s)	1.4X
SortedList.isEqual	4,4	245,025	55/145 (3m21.5s)	Reused	55/55 (24.4)	8.2X
	5,5	9,018,009	91/238 (8m52.6s)	Reused	91/91 (1m03.1s)	8.4X
	6,6	344,622,096	140/364 (25m08.0s)	Reused	140/140 (2m02.9s)	12.3X
SortedList.isIntersection	2,2,2	21,952	112/190 (2m59.1s)	Reused	112/112 (18.9s)	9.5X
	3,3,3	10,648,000	796/1006 (13m28.3s)	Reused	796/796 (3m40.0s)	3.7X
	4,4,4	6,028,568,000	Timeout[3]	Reused	5201/5201 (60m09.6s)	N/A

[1] All sizes from 0 up to this size are generated.

[2] Total non-equivalent tests are a cross product of black box tests generated by Korat [3] for each argument using an integer data domain equaling the *total* number of integers involved.

[3] Timeout is set to 5 hours.

with another field of the same type, and deleting a statement. (adapted from [20]). Six mutants of the add method are generated for each of binary search tree, sorted linked list, and binary heap.

For staged symbolic execution, we only need to run the first stage once because the pre-conditions of add method have not changed. Thus the cost of this stage is amortized over all the runs. We show our data in Table 3 and plot it as a graph in Figure 6. We call the original add method m_0 while the mutants are called m_1 – m_6. As we can see that the total time taken by staged symbolic execution is significantly less than a normal symbolic execution of all mutants. This shows the performance advantage of sharing symbolic execution results of one stage for regression testing.

4.4 Observations

Staged symbolic execution provides a number of benefits over standard symbolic execution and over test suites made using black box techniques. We observe:

Enabling symbolic test suites. The standard way to create test suites using standard symbolic execution is to concretize the tests. If this concretization is done using the class invariant alone, then we lose important tests that could have been created if the method under test was symbolically executed along with the class invariant. If however, the concretization is done after the method under test is symbolically executed, the test suite has to be regenerated whenever the method is changed. Staged symbolic execution requires a significantly smaller number of tests to be stored. It uses abstract symbolic inputs that can be saved and used to reconstruct the symbolic state at a later stage.

Performance improvements. We have seen the perfor-

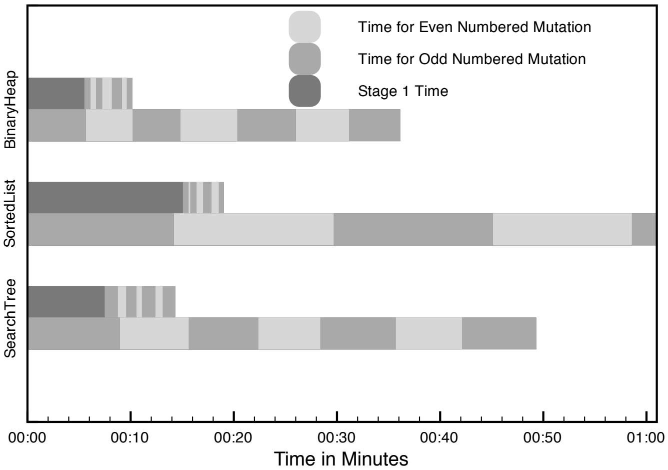

Figure 6: Comparison of time for testing the original add method and six mutants for SearchTree (size 4), SortedList (size 8), and BinaryHeap (size 10).

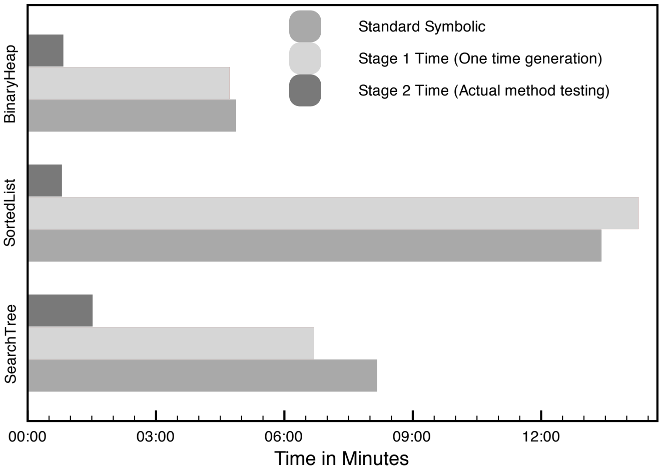

Figure 7: Comparison of time for SearchTree (size 4), SortedList (size 8), and BinaryHeap (size 10).

Table 2: Increasing number of arguments.

# args	Total non-equiv. tests[1]	Symbolic Exec. Valid/Explored tests	Symbolic Exec. (Time)	Staged Symbolic Exec. Valid/Explored tests	Staged Symbolic Exec. (Time)	Savings
SearchTree.union (size 2)						
2	625	16/96	(47.7s)	20/36	(11.6s)	4.1X
3	117,649	64/400	(14m18.9s)	68/84	(1m53.5s)	7.6X
4	43,046,721	256/1616	(244m23.5s)	260/276	(30m44.8s)	7.9X
SortedList.union (size 2)						
2	225	9/33	(9.7s)	12/18	(7.2s)	1.3X
3	21,952	27/105	(1m29.3s)	30/36	(19.0s)	4.7X
4	4,100,625	81/321	(10m50.9s)	84/90	(1m47.1s)	6.1X

[1] Total non-equivalent tests calculated as in Table 1.

Table 3: Comparison of time for regression testing.

benchmark		Symbolic Exec. Valid/Explored Tests (Time)	Staged Symbolic Execution Stage 1 Valid/Exp. (Time)	Staged Symbolic Execution Stage 2 Valid/Explored Tests (Time)
Mutants of SearchTree .add (size 4)	m_0	99/344 (7m54.7s)	23/268 (6m26.5s)	99/99 (1m16.1s)
	m_1	45/290 (6m41.5s)		45/45 (49.7s)
	m_2	71/316 (6m43.5s)		71/71 (58.3s)
	m_3	64/309 (6m01.4s)		64/46 (33.2s)
	m_4	64/309 (7m18.3s)		64/64 (1m17.3s)
	m_5	64/309 (6m25.7s)		64/64 (44.0s)
	m_6	99/344 (7m12.7s)		99/99 (1m13.0s)
		Total: 48m17.8s	**Total:** 13m18.1s **Savings:** 3.6X	
Mutants of SortedList.add .add (size 8)	m_0	45/90 (13m09.3s)	9/54 (14m01.6s)	45/45 (33.1s)
	m_1	17/62 (15m30.0s)		17/17 (10.6s)
	m_2	24/69 (15m25.7s)		24/24 (34.6s)
	m_3	45/90 (13m28.2s)		45/45 (39.9s)
	m_4	45/90 (14m51.6s)		45/45 (47.1s)
	m_5	45/90 (15m27.4s)		45/45 (44.2s)
	m_6	45/90 (14m50.5s)		45/45 (39.0s)
		Total: 102m42.7s	**Total:** 18m10.1s **Savings:** 5.7X	
Mutants of BinaryHeap .add (size 10)	m_0	32/198 (4m37.0s)	11/177 (4m28.2s)	32/32 (35.1s)
	m_1	24/190 (4m32.9s)		24/24 (33.4s)
	m_2	28/194 (4m37.9s)		28/28 (35.3s)
	m_3	24/190 (5m30.6s)		24/24 (57.0s)
	m_4	24/190 (5m40.6s)		24/24 (57.7s)
	m_5	13/179 (5m09.2s)		13/13 (29.4s)
	m_6	25/191 (4m57.9s)		25/25 (32.3s)
		Total: 35m06.1s	**Total:** 9m08.4s **Savings:** 3.8X	

mance benefit of staged symbolic execution in Table 1. We plot some data from that table in Figure 7. We can see that Stage 1 and standard symbolic execution take more or less the same amount of time, whereas Stage 2 takes significantly less time than a complete run of symbolic execution.

Library of abstract symbolic tests. We have seen that we have isolated most of the overhead of symbolic execution for structurally complex inputs in the first stage of staged symbolic execution. This stage works regardless of the method under test. Also, the number of generated tests is manageable. This means, we can create a library of abstract symbolic tests. When an abstract symbolic test from such a library is used, there is no overhead at all and the first invocation gets all the benefit. Such a library can contain abstract symbolic tests for common structures like lists, trees, etc. The library can even be generated using other tools, e.g. black box tools like Alloy or Korat.

5. RELATED WORK

Clarke [8] and King [18] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [4] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [17] shows how to apply traditional symbolic execution to object-oriented code and uses *lazy initialization* to handle pointer aliasing.

Symbolic execution guided by concrete inputs is one of the widely studied approaches for systematic bug finding in the last five years. DART [14] combines concrete and symbolic execution to collect branch conditions along the execution path. It negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers. To overcome path explosion in large programs, SMART [13] introduced inter-procedural static analysis to compute procedure summaries and reduce the paths to be explored by DART; SMART's procedure summaries bear resemblance to abstract symbolic tests but serve a very different purpose — summaries allow symbolic excution to avoid following method calls, whereas abstract symbolic tests are expanded into concrete tests as required during the second stage of staged symbolic execution. CUTE [21] extends DART to handle constraints on references.

EGT [5] and EXE [6] also use negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. KLEE [7] is the most recent tool in this family. It is open-sourced and has been used by a variety of users in academia and industry. KLEE works on LLVM byte code [1] of unmodified C/C++ programs and has been shown to work for many off the shelf programs. Staged symbolic execution uses KLEE as an enabling technology.

All the above approaches require verifying the input preconditions and running the program under test together. Thus there is no concept of storing symbolic tests as part of test suites. The only tests that can be stored are concrete tests that are generated after analysis of the program under test. These tests are only useful as long as the program under test does not change. This is in contrast to black box test suites which are generated based on input pre-conditions and do not depend on the program under test.

Alloy [16] and Korat [3] can be used for structurally complex input generation in a black box manner. Alloy is a first-order declarative logic based on sets and relations, and is supported by its fully automatic, SAT-based analyzer. TestEra [19] uses the Alloy tool-set for test input generation. Korat is a solver for structural constraints written as Java predicates. It takes an input bound specification and a predicate written in an imperative language (e.g. Java). It performs an efficient search within the input bounds and finds all instances where the predicate returns `true`.

Using Korat or TestEra to generate a test suite results in a very large number of tests because they perform black-box testing. Also, using Korat or TestEra, we have to give the bounds for every object field since they only generate concrete tests. On the other hand, staged symbolic execution depends on symbolic constraints and does not require a priori concrete bounds on all fields. Whispec [22] builds on TestEra and solves method pre-conditions written in Alloy but uses a form of dynamic symbolic execution to guide concrete test generation for increasing code coverage.

Abstract subsumption checking [2] presents an approach for testing an under-approximation of the program using symbolic execution. It introduces abstractions for lists and arrays and checks if a symbolic state is subsumed by an earlier state under the abstractions. If so, the search is backtracked. This way symbolic execution can substantially reduce the number of explored paths for the program under-approximated using the abstractions.

6. CONCLUSIONS

We presented a novel approach to increase the efficiency of symbolic execution for systematic testing of object-oriented programs. We applied symbolic execution in *stages*, rather than the traditional approach of applying it all at once, to compute *abstract symbolic inputs* that are shared across different methods to test them systematically. We used class invariants to first generate abstract symbolic tests, which are then used to symbolically execute several methods that require their inputs to satisfy the invariant. We presented an experimental evaluation and comparison against KLEE, a state-of-the-art implementation of symbolic execution. Results show that our approach enables significant savings in the cost of systematic testing using symbolic execution.

Acknowledgements

We thank Darko Marinov for detailed comments on an earlier draft of this paper. This work was funded in part by the Fulbright Program, the NSF under Grant Nos. CCF-0845628 and IIS-0438967, and AFOSR grant FA9550-09-1-0351.

7. REFERENCES

[1] V. Adve et al. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proc. MICRO-36*, 2003.

[2] S. Anand et al. Symbolic Execution with Abstraction. *Int. J. Softw. Tools Technol. Transf.*, 11, 2009.

[3] C. Boyapati et al. Korat: Automated Testing based on Java Predicates. In *Proc. ISSTA*, 2002.

[4] W. R. Bush et al. A Static Analyzer for Finding Dynamic Programming Errors. *Softw. Pract. Exper.*, 30(7), 2000.

[5] C. Cadar and D. Engler. Execution Generated Test Cases: How to make systems code crash itself. In *Proc. SPIN*, 2005.

[6] C. Cadar et al. EXE: Automatically Generating Inputs of Death. In *Proc. CCS*, 2006.

[7] C. Cadar et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*, 2008.

[8] L. A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[9] B. Daniel et al. Automated Testing of Refactoring Engines. In *Proc. ESEC/FSE*, 2007.

[10] H. Do and G. Rothermel. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Softw. Eng.*, 32, 2006.

[11] J. P. Galeotti et al. Analysis of Invariants for Efficient Bounded Verification. In *Proc. ISSTA*, 2010.

[12] M. Gligoric et al. Test Generation through Programming in UDITA. In *Proc. ICSE*, 2010.

[13] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. POPL*, 2007.

[14] P. Godefroid et al. DART: Directed Automated Random Testing. In *Proc. PLDI*, 2005.

[15] P. Godefroid et al. Automated Whitebox Fuzz Testing. In *Proc. NDSS*, 2008.

[16] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[17] S. Khurshid et al. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. TACAS*, 2003.

[18] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), 1976.

[19] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proc. ASE*, 2001.

[20] J. Offutt et al. An Experimental Mutation System for Java. *SIGSOFT Softw. Eng. Notes*, 29(5), 2004.

[21] K. Sen et al. CUTE: A Concolic Unit Testing Engine for C. In *Proc. ESEC/FSE*, 2005.

[22] D. Shao et al. Whispec: White-box Testing of Libraries using Declarative Specifications. In *Proc. LCSD*, 2007.

[23] K. Sullivan et al. Software Assurance by Bounded Exhaustive Testing. In *Proc. ISSTA*, 2004.

[24] N. Tillmann and J. De Halleux. Pex: White box Test Generation for .NET. In *Proc. TAP*, 2008.

[25] W. Visser et al. Model Checking Programs. *Automated Softw. Eng. J.*, 10(2), 2003.