Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution

Tao Xie[1], Darko Marinov[2], Wolfram Schulte[3], and David Notkin[1]

[1] Dept. of Computer Science & Engineering, Univ. of Washington, Seattle, WA 98195, USA
[2] Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801, USA
[3] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{taoxie, notkin}@cs.washington.edu, marinov@cs.uiuc.edu,
schulte@microsoft.com

Abstract. Object-oriented unit tests consist of sequences of method invocations. Behavior of an invocation depends on the method's arguments and the state of the receiver at the beginning of the invocation. Correspondingly, generating unit tests involves two tasks: generating method sequences that build relevant receiver-object states and generating relevant method arguments. This paper proposes Symstra, a framework that achieves both test generation tasks using symbolic execution of method sequences with symbolic arguments. The paper defines symbolic states of object-oriented programs and novel comparisons of states. Given a set of methods from the class under test and a bound on the length of sequences, Symstra systematically explores the object-state space of the class and prunes this exploration based on the state comparisons. Experimental results show that Symstra generates unit tests that achieve higher branch coverage faster than the existing test-generation techniques based on concrete method arguments.

1 Introduction

Object-oriented unit tests are programs that test classes. Each test case consists of a fixed sequence of method invocations with fixed arguments that explores a particular aspect of the behavior of the class under test. Unit tests are becoming an important component of software development. The Extreme Programming discipline [5], for instance, leverages unit tests to permit continuous and controlled code changes. Unlike in traditional testing, it is developers (not testers) who write tests for every aspect of the classes they develop. However, manual test generation is time consuming, and so typical unit test suites cover only some aspects of the class.

Since unit tests are gaining importance, many companies now provide tools, frameworks, and services around unit tests. Tools range from specialized test frameworks, such as JUnit [18] or Visual Studio's new team server [25], to automatic unit-test generation, such as Parasoft's Jtest [27]. However, existing test-generation tools typically do not provide guarantees about the generated unit-test suites. In particular, the suites rarely satisfy the branch-coverage test criterion [6], let alone a stronger criterion, such as the bounded intra-method path coverage [3] of the class under test. We present an approach that uses symbolic execution to exhaustively explore bounded method sequences of the

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 365–381, 2005.

class under test and to generate tests that achieve high branch and intra-method path coverage for complex data structures such as container implementations.

1.1 Background

Generating test sequences involves two tasks: generating method sequences that build relevant receiver-object state and generating relevant method arguments. Researchers have addressed this problem several times. Most tools generate test sequences using concrete representations. A popular approach is to use (smart) random generation; this approach is embodied in tools such as Jtest [27] (a commercial tool for Java) or JCrasher [13] and Eclat [26] (two research prototypes for Java). Random tests generated by these tools often execute the same sequences [34] and are not covering (do not cover all sequences). The AsmLT model-based testing tool [16, 15] uses concrete-state space-exploration techniques [12] to generate covering method sequences. But AsmLT requires the user to carefully choose sufficiently large concrete domains for method arguments and the right abstraction functions to guarantee the covering. Tools such as Korat [8] are able to generate non-isomorphic object graphs that can be used for testing, but they do not generate covering test sequences.

King proposed in the 70's to use symbolic execution for testing and verification [20]. Because of the advances in constraint solvers, this technique recently regained the attention for test generation. For example, the BZ-TT tool uses constraint solving to derive method sequences from B specifications [22]. However, the B specifications are not object-oriented. Khurshid et al. [19, 33] proposed an approach for generating tests for Java classes based on symbolic execution. They show that their generation based on symbolic execution generates tests faster than their model checking of method sequences with concrete arguments. This is expected: symbolic representations describe not only single states, but sets of states, and when applicable, symbolic representations can yield large improvements, witnessed for example by symbolic model checking [24]. The approach of Khurshid et al. [19, 33], however, generates the receiver-object states, similar to Korat [8], only as object graphs, not through method sequences. Moreover, it requires the user to provide specially constructed class invariants [23], which effectively describe an over-approximation of the set of reachable object graphs.

Symbolic execution is the foundation of static code analysis tools. These tools typically do not generate test data, but automatically verify simple properties of programs. These properties often allow merging symbolic states that stem from different execution paths. However, for test generation, states have to be kept separate, since different tests should be used for different paths. Recently, tools such as SLAM [4, 2] and Blast [17, 7] were adapted for test generation. However, neither of them can deal with complex data structures, which are the focus of this paper.

1.2 Contributions

This paper makes the following contributions.

Symbolic Sequence Exploration: We propose Symstra, a framework that uses symbolic execution to generate method sequences. When applicable, Symstra uses an exhaustive exploration of method sequences with symbolic variables for primitive-type arguments.

(We also discuss how Symstra can handle reference-type arguments.) Each symbolic argument represents a set of all possible concrete values for the argument. Symstra uses symbolic execution to operate on symbolic states that include symbolic variables.

Symbolic State Comparison: We present novel techniques for comparison of symbolic states of object-oriented programs. Our techniques allow Symstra to prune the exploration of the object state and thus generate tests faster, without compromising the exhaustiveness of the exploration. In particular, the pruning preserves the intra-method path coverage of the generated test suites.

Implementation: We describe an implementation of a test-generation tool for Symstra. Our implementation handles dynamically allocated structures, method pre- and post-conditions, and symbolic data. Our current implementation does not support concurrency, but such support can be added by reimplementing Symstra in a Java model checker, such as Java Pathfinder [32] or Bogor [29].

Evaluation: We evaluate Symstra on seven subjects, most of which are complex data structures. The experimental results show that Symstra generates tests faster than the existing test-generation techniques based on exhaustive exploration of sequences with concrete method arguments [33, 16, 15, 34]. Further, given the same time for generation, Symstra can generate tests that achieve better branch coverage than the existing techniques. Finally, Symstra works on ordinary Java implementations and does not require the user to provide the additional methods required by some other approaches [33, 8].

2 Example

This section illustrates how Symstra explores method sequences and generates tests. Figure 1 shows a binary search tree class `BST` that implements a set of integers. Each tree has a pointer to the root node. Each node has an element and pointers to the left and right children. The class also implements the standard set operations: `insert` adds an element, if not already in the tree, to a leaf; `remove` deletes an element, if in the tree, replacing it with the smallest larger child if necessary; and `contains` checks if an element is in the tree. The class also has a default constructor that creates an empty tree.

Some tools such as Jtest [27] or JCrasher [13] test a class by generating random sequences of methods; for `BST`, they could for example generate the following tests:

```
Test 1:
  BST t1 = new BST();
  t1.insert(0);
  t1.insert(-1);
  t1.remove(0);
```

```
Test 2:
  BST t2 = new BST();
  t2.insert(2147483647);
  t2.remove(2147483647);
  t2.insert(-2147483648);
```

Each test has a method sequence on the objects of the class, e.g., Test 1 creates a tree `t1`, invokes two `insert` methods on it, and then one `remove`. Typically, checking the correctness (of outputs) for such tests relies on design-by-contract annotations translated into run-time assertions [27, 10] or on model-based testing [16]. If there are no annotations or models, the tools check only the code robustness: execute the tests and check for uncaught exceptions [13].

```
class BST implements Set {
    Node root;
    static class Node {
        int value;
        Node left;
        Node right;
    }
    void insert(int value) { ... }
    void remove(int value) { ... }
    bool contains(int value) { ... }
}
```

Fig. 1. A set implemented as a binary search tree

Some other tools [33, 16, 15, 34] can exhaustively explore all method sequences up to a given length. Such exploration raises two questions: (1) what arguments to use for method calls, and (2) how to determine equivalent tests? These tools typically require the user to provide a sufficiently good set of concrete values for each argument, or based on the argument type, use a set of default values that may miss relevant behaviors. These tools check equivalence of test sequences by comparing the states that the sequences build; the comparison uses either user-provided functions or defaults, such as identity or isomorphism. This generation is similar to explicit-state model checking [12].

Symstra also explores all sequences, but using *symbolic values* for primitive-type arguments in method calls. Such exploration relieves Symstra users from the burden of providing concrete values: Symstra determines the relevant values during the execution. Having symbolic arguments necessitates symbolic execution [20]. It operates on a symbolic state that consists of two parts: (1) a *constraint*, known as the *path condition*, that must hold for the execution to reach a certain point and (2) a heap that contains symbolic variables. When the symbolic execution encounters a branch, it explores both outcomes, appropriately adding the branch condition or its negation to the constraint. Symbolic state exploration in Symstra is conceptually similar to symbolic model checking [24].

Let us consider the symbolic execution of the following sequence:

```
BST t = new BST();
t.insert(x1);
t.insert(x2);
t.insert(x3);
t.remove(x4);
```

This sequence has four method calls whose arguments are symbolic variables x_1, x_2, x_3, and x_4. While an execution of a sequence with concrete arguments produces one state, symbolic execution of a sequence with symbolic arguments can produce several states, thus resulting in an execution tree. Figure 2 shows a part of the execution tree for this example. Each state has a heap and a constraint that must hold for that heap to be created. The constructor first creates an empty tree. The first `insert` then adds the element x_1 to the tree.

The second `insert` produces states s_3, s_4, and s_5: if $x_1 = x_2$, the tree does not change, and if $x_2 > x_1$ (or $x_2 < x_1$), x_2 is added in the right (or left) subtree. Note that the symbolic states s_2 and s_4 are *syntactically* different: s_2 has the constraint `true`, while s_4 has $x_1 = x_2$. However, these two symbolic states are *semantically* equivalent: they can be instantiated into the same set of concrete heaps by giving to x_1 and x_2

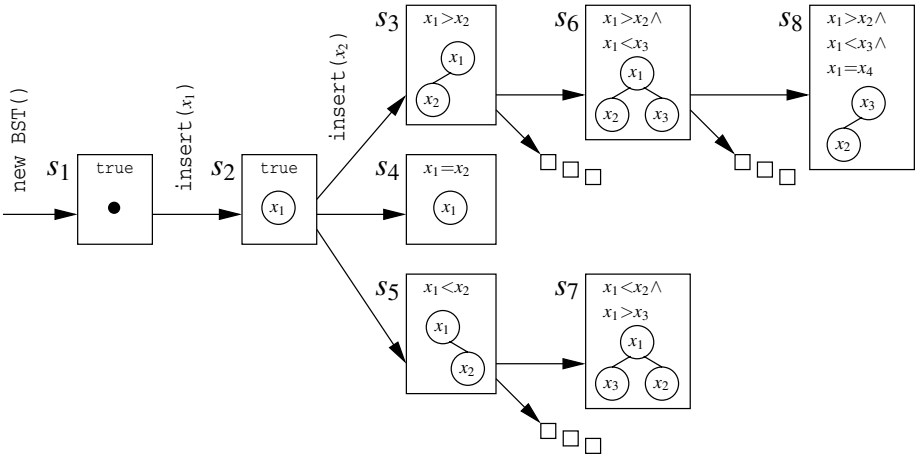

Fig. 2. A part of the symbolic execution tree

concrete values that satisfy the constraints; since x_2 does not appear in the heap in s_4, the constraint in s_4 is "irrelevant". Instead of state equivalence, it suffices to check state *subsumption*: we say that s_2 subsumes s_4 because the set of concrete heaps of s_4 is a subset of the set of concrete heaps of s_2. Hence, Symstra does not need to explore s_4 after it has already explored s_2. Symstra detects this by checking that the implication of constraints $x_1 = x_2 \Rightarrow \texttt{true}$ holds. Our current Symstra implementation uses the Omega library [28] and CVC Lite [11] to check the validity of the implication.

The third `insert` again produces several symbolic states. Symstra applies `insert` only on s_3 and s_5 (and not on s_4). In particular, we focus on s_6 and s_7, two of the symbolic states that these executions produce. These two states are syntactically different, but semantically equivalent: we can exchange the variables x_2 and x_3 to obtain the same symbolic state. Symstra detects this by checking that s_6 and s_7 are *isomorphic* (Section 3.2). Symstra finally applies `remove`. Note again that one of the symbolic states produced, s_8, is subsumed by a previously explored state, s_3.

This example has illustrated how Symstra would explore symbolic execution for one particular sequence. Symstra actually exhaustively explores the symbolic execution tree for all sequences up to a given length, pruning the exploration based on subsumption. These sequences consists of all specified methods of the class under test, i.e., `insert`, `remove`, and `contains` for `BST`.

After producing a symbolic state s, Symstra can generate a specific test with concrete arguments to produce a concrete heap of s. Symstra generates the test by traversing the shortest path from the root of the symbolic execution tree to s and outputting the method calls that it encounters. To generate concrete arguments for these calls, Symstra uses a constraint solver. Our current implementation uses the POOC solver [31]. For example, the tests that it generates for s_3 and s_5 are:

```
Test for s3:
  BST t3 = new BST();
  t3.insert(-999999);
  t3.insert(-1000000);
```

```
Test for s5:
     BST t5 = new BST();
     t5.insert(-1000000);
     t5.insert(-999999);
```

A realistic suite of unit tests contains more sequences that test the interplay between `insert`, `remove`, and `contains` methods. Section 4 summarizes such suites.

3 Framework and Implementation

We next formalize the notions introduced informally in the previous section. We first describe how Symstra represents symbolic states. Symstra uses them for two purposes: (1) during the symbolic execution of method invocations and (2) for representing the states between method invocations in method sequences. We then present how Symstra compares states based on the isomorphism of heaps and implication of constraints. We next present the symbolic execution of method invocations. We finally present the systematic exploration of method sequences and how Symstra uses symbolic state comparison to prune this exploration. We present the Symstra technique itself as well as our current implementation.

3.1 Symbolic State

Symbolic states differ from concrete states, on which the usual program executions operate, in that symbolic states contain symbolic expressions with symbolic variables and also constraints on these variables [20]. Symstra uses the following symbolic expressions and constraints:

- A symbolic variable is a symbolic expression. Each symbolic variable has a type, which is one of the Java types. For example, x_1 and x_2 may be each a symbolic variable (and thus also a symbolic expression) of type `int`.
- A Java constant of some type is a symbolic expression of that type.
- For each Java operator $\odot$ with n operands, n symbolic expressions of the appropriate operand types connected with $\odot$ are a symbolic expression of the result type. For example, $x_1 + x_2$ and $x_1 > x_2$ are expressions of type `int` and `boolean`, respectively.
- Symbolic expressions of type `boolean` are constraints.

Let P be the set of all primitive values, including integers, `true`, `false`, etc. Let V be a set of infinite number of symbolic variables of each type and U a set of all possible expressions formed from V and P. Given a valuation for the variables, $\eta : V \to P$, we extend it to evaluate all expressions $\eta : U \to P$ as follows: $\eta(p) = p$ for all $p \in P$, and $\eta(\odot u_1, \ldots, u_n) = \text{eval}(\odot, \eta(u_1), \ldots, \eta(u_n))$ for all $u_1, \ldots, u_n \in U$ and operations $\odot$, where eval evaluates operations on primitive values according to the Java semantics.

In object-oriented programs, a concrete state consists of a global heap and a stack (in general one stack for each thread, but we consider here only single-threaded programs), as well as several other parts, such as metadata for classes and program counters. Symbolic states in Symstra have the same parts as concrete states, but the heaps and stacks in

symbolic states can contain symbolic expressions; additionally, each symbolic state has a constraint. We focus on the symbolic state between method sequences.

Definition 1. *A* symbolic state $\langle C, H\rangle$ *is a pair of a constraint and a symbolic heap.*

We view each heap as a graph: nodes represent objects (as well as primitive values and symbolic expressions) and edges represent object fields. Let O be some set of objects whose fields form a set F. Each object has a field that represents its class. We consider arrays as objects whose fields are labelled with (integer) array indexes and point to the array elements.

Definition 2. *A symbolic heap is an edge-labelled graph* $\langle O, E\rangle$*, where* $E \subseteq O \times F \times (O \cup \{\texttt{null}\} \cup U)$ *such that for each field* f *of each* $o \in O$ *exactly one* $\langle o, f, o'\rangle \in E$*. A concrete heap has only concrete values:* $o' \in O \cup \{\texttt{null}\} \cup P$*.*

3.2 Heap Isomorphism

We define heap isomorphism as graph isomorphism based on node bijection [8]. We are interested in detecting isomorphic heaps because they lead to equivalent method behaviors; hence, it suffices to explore only one representative from each isomorphism partition. Nodes in symbolic heaps contain symbolic variables, so we first define a renaming of symbolic variables. Given a bijection $\tau : V \to V$, we extend it to the whole $\tau : U \to U$ as follows: $\tau(p) = p$ for all $p \in P$, and $\tau(\odot u_1, \ldots, u_n) = \odot\tau(u_1), \ldots, \tau(u_n)$ for all $u_1, \ldots, u_n \in U$ and operations $\odot$. We further extend τ to substitute free variables in formulas with bound variables, avoiding capture as usual.

Definition 3. *Two heaps* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there are bijections* $\rho : O_1 \to O_2$ *and* $\tau : V \to V$ *such that:*

$$E_2 = \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \{\langle \rho(o), f, \texttt{null}\rangle | \langle o, f, \texttt{null}\rangle \in E_1\} \cup \{\langle \rho(o), f, \tau(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in U\}.$$

Note that the definition allows only object identities and symbolic variables to vary: two isomorphic heaps have the same fields for all objects and equal (up to renaming) symbolic expressions for all primitive fields.

The state exploration in Symstra focuses on the state of several objects and does not consider the entire heap; in this context, the state of an object o consists of the values of the fields of o and fields of all objects *reachable* from o. From a program heap $\langle O, E\rangle$ and a tuple $\langle v_0, \ldots, v_n\rangle$ of pointers and symbolic expressions $v_i \in O \cup U$, where $0 \le i \le n$, Symstra constructs a *rooted heap* [34] $\langle O_h, E_h\rangle$ that has a unique root object $r \in O_h$: Symstra first creates the heap $\langle O', E'\rangle$, where $O' = O \cup \{r\}$, $r \notin O$, and $E' = E \cup \{\langle r, i, v_i\rangle | 0 \le i \le n\}$, and then creates $\langle O_h, E_h\rangle$ as the subgraph of $\langle O', E'\rangle$ such that $O_h \subseteq O'$ is the set of all objects reachable from r within E' and $E_h = \{\langle o, f, o'\rangle \in E' | o \in O_h\}$.

We can efficiently check isomorphism of rooted heaps, even though for general graphs it is unknown whether checking isomorphism can be done in polynomial time. Symstra *linearizes* heaps into integer sequences such that checking heap isomorphism corresponds to checking sequence equality. Figure 3 shows the linearization algorithm.

```
Map<Object,int> objs; // maps objects to unique ids
Map<SymVar,int> vars; // maps symbolic variables to unique ids

int[] linearize(Object root, Heap <O,E>) {
  objs = new Map();  vars = new Map();
  return lin(root, <O,E>>;
}

int[] lin(Object root, Heap <O,E>) {
  if (objs.containsKey(root))
    return singletonSequence(objs.get(root));
  int id = objs.size() + 1;  objs.put(root, id);
  int[] seq = singletonSequence(id);
  Edge[] fields = sortByField({ <root, f, o> in E });
  foreach (<root, f, o> in fields) {
    if (isSymbolicExpression(o)) seq.append(linSymExp(o));
    elseif (o == null) seq.append(0);
    else seq.append(lin(o, <O,E>)); // pointer to an object
  }
  return seq;
}

int[] linSymExp(SymExp e) {
  if (isSymVar(e)) {
    if (!vars.containsKey(e))
      vars.put(e, vars.size() + 1);
    return singletonSequence(vars.get(e));
  } elseif (isPrimitive(e)) return uniqueRepresentation(e);
  else { // operation with operands
    int[] seq = singletonSequence(uniqueRepresentation(e.getOperation()));
    foreach (SymExp e' in e.getOperands())
      seq.append(linSymExp(e'));
    return seq;
  }
}
```

Fig. 3. Pseudo-code of linearization for a symbolic rooted heap

It starts from the root and traverses the heap depth first. It assigns a unique identifier to each object, keeps this mapping in `objs` and reuses it for objects that appear in cycles. It also assigns a unique identifier to each symbolic variable, keeps this mapping in `vars` and reuses it for variables that appear several times in the heap.

A similar linearization is used to represent concrete heaps in model checking [1,30, 32]. This paper extends the linearization from our previous work [34] with `linSymExp` that handles symbolic expressions; this improves on the approach of Khurshid et al. [19, 33] that does not use any comparison for symbolic expressions. It is easy to show that our linearization normalizes rooted heaps.

Theorem 1. *Two rooted heaps* $\langle O_1, E_1\rangle$ *(with root* r_1*) and* $\langle O_2, E_2\rangle$ *(with root* r_2*) are isomorphic iff* $\texttt{linearize}(r_1, \langle O_1, E_1\rangle) = \texttt{linearize}(r_2, \langle O_2, E_2\rangle)$.

3.3 State Subsumption

We define symbolic state subsumption based on the concrete heaps that each symbolic state represents. Symstra uses state subsumption to prune the exploration. To instantiate

```
boolean checkSubsumes(Constraint C1, Heap H1,
                      Constraint C2, Heap H2) {
    int[] i1 = linearize(root(H1), H1);
    Map<SymVar,int> v1 = vars; // at the end of previous linearization
    Set<SymVar> n1 = variables(C1) - v1.keys(); // variables not in the heap
    int[] i2 = linearize(root(H2), H2);
    Map<SymVar,int> v2 = vars; // at the end of previous linearization
    Set<SymVar> n2 = variables(C2) - v2.keys(); // variables not in the heap
    if (i1 <> i2) return false;
    Renaming τ = v2 o v1^-1 // compose v2 and the inverse of v1
    return checkValidity(τ(∃n2. C2) ⇒ ∃n1. C1);
}
```

Fig. 4. Pseudo-code of subsumption checking for symbolic states

a symbolic heap into a concrete heap, we replace the symbolic variables in the heap with primitive values that satisfy the constraint in the symbolic state.

Definition 4. *An* instantiation $\mathcal{I}(\langle C, H\rangle)$ *of a symbolic state* $\langle C, H\rangle$ *is a set of concrete heaps* H' *such that there exists a valuation* $\eta : V \rightarrow P$ *for which* $\eta(C)$ *is true and* H' *is the evaluation* $\eta(H)$ *of all expressions in* H *according to* η*.*

Definition 5. *A symbolic state* $\langle C_1, H_1\rangle$ subsumes *another symbolic state* $\langle C_2, H_2\rangle$*, in notation* $\langle C_1, H_1\rangle \supseteq \langle C_2, H_2\rangle$*, iff for each concrete heap* $H'_2 \in \mathcal{I}(\langle C_2, H_2\rangle)$*, there exists a concrete heap* $H'_1 \in \mathcal{I}(\langle C_1, H_1\rangle)$ *such that* H'_1 *and* H'_2 *are isomorphic.*

Symstra uses the algorithm in Figure 4 to check if the constraint of $\langle C_1, H_1\rangle$, after suitable renaming, implies the constraint of $\langle C_2, H_2\rangle$. Note that the implication is universally quantified over the (renamed) symbolic variables that appear in the heaps and existentially quantified over the symbolic variables that do not appear in the heaps (more precisely only in H_1, because the existential quantifier for n_2 in the premise of the implication becomes a universal quantifier for the whole implication). We can show that this algorithm is a conservative approximation of subsumption.

Theorem 2. *If* `checkSubsumes`$(\langle C_1, H_1\rangle, \langle C_2, H_2\rangle)$ *then* $\langle C_1, H_1\rangle$ *subsumes* $\langle C_2, H_2\rangle$*.*

Symstra gains the power and inherits the limitations from the technique used to check the implication on the (renamed) constraints. The current Symstra prototype uses the Omega library [28], which provides a complete decision procedure for Presburger arithmetic, and CVC Lite [11], an automatic theorem prover, which has decision procedures for several types of constraints, including real linear arithmetic, uninterpreted functions, arrays, etc. Since these checks can consume a lot of time, Symstra further uses the following conservative approximation: if free-variables($\exists n_1.\ C_1$) are not a subset of free-variables($\tau(\exists n_2.\ C_2)$), return `false` without checking the implication.

3.4 Symbolic Execution

We next discuss the symbolic execution of one method in a method sequence. Each method execution starts with one symbolic state and produces several symbolic states. We use the notation $\sigma_m(\langle C, H\rangle)$ to denote the set $\{\langle C_1, H_1\rangle, \ldots, \langle C_n, H_n\rangle\}$ of states that the symbolic execution, σ, of the method m produces starting from the state $\langle C, H\rangle$.

Following the typical symbolic executions [20, 33], Symstra symbolically explores both branches of `if` statements, modifying the constraint with a conjunct that needs to hold for the execution to take a certain branch. In this context, the constraint is called *path condition*, because it is a conjunction of conditions that need to hold for the execution to take a certain path and reach the current address. This symbolic execution directly explores every path of the method under consideration. The common issue in the symbolic execution is that the number of paths may be infinite (or too large as it grows exponentially with the number of branches) and thus $\sigma_m(\langle C, H\rangle)$ may be (practically) unbounded. In such cases, Symstra can use the standard set of heuristics to explore only some of the paths [33, 9].

The current Symstra prototype implements the execution steps on symbolic state by rewriting the code to operate on symbolic expressions. Further, Symstra implements the exploration of different branches by re-executing the method from the beginning for each path, without storing any intermediate states. Note that Symstra re-executes only one method (for different paths), not the whole method sequence. (This effectively produces a depth-first exploration of paths within one method, while the exploration of states between methods is breadth-first as explained in the next section.)

Our Symstra prototype also implements the standard optimizations for symbolic execution. First, Symstra simplifies the constraints that it builds at branches; specifically, before conjoining the path condition so far C and the current branch condition C' (where C' is a condition from an `if` or its negation), Symstra checks if some of the conjuncts in C implies C'; if so, Symstra does not conjoin C'. Second, Symstra checks if the constraint $C\&\&C'$ is unsatisfiable; if so, Symstra stops the current path of symbolic execution, because it is an infeasible path. The current Symstra prototype can use the Simplify [14] theorem prover or the Omega library [28] to check unsatisfiability. We have found that Omega is faster, but it handles only linear arithmetic constraints.

Given a symbolic state at the entry of a method execution, Symstra uses symbolic execution to achieve structural coverage within the method, because symbolic execution systematically explores all feasible paths within the method. If the user of Symstra is interested in only the tests that achieve new branch coverage, our Symstra prototype monitors the branch coverage during symbolic execution and selects a symbolic execution for concrete test generation (Section 3.6) when the symbolic execution covers a new branch. The Symstra prototype can also be extended for selecting symbolic executions that achieve new bounded intra-method path coverage [3].

3.5 Symbolic State Exploration

We next present the symbolic state space for method sequences and how Symstra systematically explores this state space. The state space consists of all states that are reachable with the symbolic execution of all possible method sequences for the class under test. Let $\mathcal{C}$ and $\mathcal{M}$ be a set of the constructor and non-constructor methods of this class. Each method sequence starts with a constructor from $\mathcal{C}$ followed by several methods from $\mathcal{M}$. We denote with $\Sigma_{\mathcal{C},\mathcal{M}}$ the state space for these sequences. The initial symbolic state is $s_0 = \langle \mathtt{true}, \{\}\rangle$: the constraint is true, and the heap is empty. The state space includes the states that the symbolic execution produces for the constructors and

methods: $\bigcup_{c \in \mathcal{C}} \sigma_c(s_0) \subset \Sigma_{\mathcal{C},\mathcal{M}}$ and $\forall s \in \Sigma_{\mathcal{C},\mathcal{M}}. \bigcup_{m \in \mathcal{M}} \sigma_m(s) \subset \Sigma_{\mathcal{C},\mathcal{M}}$. As usual [12], $\Sigma_{\mathcal{C},\mathcal{M}}$ is the least fixed point of these equations. The state space is typically infinite.

The current Symstra prototype exhaustively explores a bounded part of the symbolic state space using a breadth-first search. The inputs to Symstra are a set of constructor $\mathcal{C}$ and non-constructor methods $\mathcal{M}$ of the class under test and a bound on the length of sequences. Symstra maintains a set of explored states and a processing queue of states. Symstra processes the queue in a breadth-first manner: it takes one state and symbolically executes each method under test (constructor at the beginning of the sequence and a non-constructor after that) for each path on this state. Every such execution yields a new symbolic state. Symstra adds the new state to the queue for further exploration only if it is not subsumed by an already explored state from the set. Otherwise, Symstra prunes the exploration: the new symbolic state represents only a subset of the concrete heaps that some explored symbolic state represents; it is thus unnecessary to explore the new state further. Pruning based on subsumption plays the key role in enabling Symstra to explore large state spaces.

3.6 Concrete Test Generation

During the symbolic state exploration, Symstra also builds specific concrete tests that lead to the explored states. Whenever Symstra finishes a symbolic execution of a method that generates a new symbolic state $\langle C, H \rangle$, it also generates a *symbolic test*. This test consists of the constraint C and the shortest method sequence that reaches $\langle C, H \rangle$. (Symstra associates such a method sequence with each symbolic state and dynamically updates it during execution). Symstra then instantiates a symbolic test using the POOC constraint solver [31] to solve the constraint C over the symbolic arguments for methods in the sequence. Based on the produced solution, Symstra obtains concrete arguments for the sequence leading to $\langle C, H \rangle$. Symstra exports such concrete test sequences into a JUnit test class [18]. It also exports the constraint C associated with the test as a comment for the test in the JUnit test class.

At the class-loading time, Symstra instruments each branching point of the class under test for measuring branch coverage at the bytecode level. It also instruments each method of the class to capture uncaught exceptions at runtime. The user can configure Symstra to select only those generated tests that increase branch coverage or throw new uncaught exceptions.

4 Evaluation

This section presents our evaluation of Symstra for exploring method sequences and generating tests. We compare Symstra with Rostra [34], our previous framework that generates tests using bounded-exhaustive exploration of sequences with concrete arguments. We have developed Symstra on top of Rostra, so that the comparison does not give an unfair advantage to Symstra because of unrelated improvements. In these experiments, we have used the Simplify [14] theorem prover to check unsatisfiability of path conditions, the Omega library [28] to check implications, and the POOC constraint solver [31] to solve constraints. We have performed the experiments on a Linux machine

Table 1. Experimental subjects

class	methods under test	some private methods	#ncnb lines	# branches
IntStack	push,pop	–	30	9
UBStack	push,pop	–	59	13
BinSearchTree	insert,remove	removeNode	91	34
BinomialHeap	insert,extractMin delete	findMin,merge unionNodes,decrease	309	70
LinkedList	add,remove,removeLast	addBefore	253	12
TreeMap	put,remove	fixAfterIns fixAfterDel,delEntry	370	170
HeapArray	insert,extractMax	heapifyUp,heapifyDown	71	29

with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory.

Table 1 lists the seven Java classes that we use in the experiments. The first six classes were previously used in evaluating Rostra [34], and the last five classes were used in evaluating Korat [8]. The columns of the table show the class name, the public methods under test (that the generated sequences consist of), some private methods invoked by the public methods, the number of non-comment, non-blank lines of code in all those methods, and the number of branches for each subject.

We use Symstra and Rostra to generate test sequences with up to N methods. Rostra also requires concrete values for arguments, so we set it to use N different arguments (the integers from 0 to $N-1$) for methods under test. Table 2 shows the comparison between Symstra and Rostra. We range N from five to eight. (For $N < 5$, both Symstra and Rostra generate tests really fast, usually within a couple of seconds, but those tests do not have good quality.) We tabulate the time to generate the tests (measured in seconds, Columns 3 and 7), the number of explored symbolic and concrete object states (Columns 4 and 8), the number of generated tests (Columns 5 and 9), and the branch coverage[1] achieved by the generated tests (Columns 6 and 10). In Columns 5 and 9, we report the total number of generated tests and, in the parentheses, the cumulative number of tests that increase the branch coverage.

During test generation, we set a three-minute timeout for each iteration of the breadth-first exploration: when an iteration exceeds three minutes, the exhaustive exploration of Symstra or Rostra is stopped and the system proceeds with the next iteration. We use a "*" mark for each entry where the test-generation process timed out; the state exploration of these entries is no longer exhaustive. We use a "–" mark for each entry where Symstra or Rostra exceeded the memory limit.

The results indicate that Symstra generates method sequences of the same length N often much faster than Rostra, thus enabling Symstra to generate longer method sequences within a given time limit. Both Symstra and Rostra achieve the same branch

[1] We measure the branch coverage at the bytecode level during the state exploration of both Symstra and Rostra, and calculate the total number of branches also at the bytecode level.

Table 2. Experimental results of test generation using Symstra and Rostra

		Symstra				Rostra			
class	N	time	states	tests	%cov	time	states	tests	%cov
UBStack	5	0.95	22	43(5)	92.3	4.98	656	1950(6)	92.3
	6	4.38	30	67(6)	100.0	31.83	3235	13734(7)	100.0
	7	7.20	41	91(6)	100.0	*269.68	*10735	*54176(7)	*100.0
	8	10.64	55	124(6)	100.0	-	-	-	-
IntStack	5	0.23	12	18(3)	55.6	12.76	4836	5766(4)	55.6
	6	0.42	16	24(4)	66.7	-	-	-	-
	7	0.50	20	32(5)	88.9	*689.02	*30080	*52480(5)	*66.7
	8	0.62	24	40(6)	100.0	-	-	-	-
BinSearchTree	5	7.06	65	350(15)	97.1	4.80	188	1460(16)	97.1
	6	28.53	197	1274(16)	100.0	23.05	731	7188(17)	100.0
	7	136.82	626	4706(16)	100.0	-	-	-	-
	8	*317.76	*1458	*8696(16)	*100.0	-	-	-	-
BinomialHeap	5	1.39	6	40(13)	84.3	4.97	380	1320(12)	84.3
	6	2.55	7	66(13)	84.3	50.92	3036	12168(12)	84.3
	7	3.80	8	86(15)	90.0	-	-	-	-
	8	8.85	9	157(16)	91.4	-	-	-	-
LinkedList	5	0.56	6	25(5)	100.0	32.61	3906	8591(6)	100.0
	6	0.66	7	33(5)	100.0	*412.00	*9331	*20215(6)	*100.0
	7	0.78	8	42(5)	100.0	-	-	-	-
	8	0.95	9	52(5)	100.0	-	-	-	-
TreeMap	5	3.20	16	114(29)	76.5	3.52	72	560(31)	76.5
	6	7.78	28	260(35)	82.9	12.42	185	2076(37)	82.9
	7	19.45	59	572(37)	84.1	41.89	537	6580(39)	84.1
	8	63.21	111	1486(37)	84.1	-	-	-	-
HeapArray	5	1.36	14	36(9)	75.9	3.75	664	1296(10)	75.9
	6	2.59	20	65(11)	89.7	-	-	-	-
	7	4.78	35	109(13)	100.0	-	-	-	-
	8	11.20	54	220(13)	100.0	-	-	-	-

coverage for method sequences of the same length N. However, Symstra achieves higher coverage faster. It also takes less memory and can finish generation in more cases than Rostra. These results are due to the fact that each symbolic state, which Symstra explores at once, actually describes a set of concrete states, which Rostra must explore one by one. Rostra often exceeds the memory limit when $N = 7$ or $N = 8$, which is often not enough to guarantee full branch coverage.

5 Discussion and Future Work

Specifications. Symstra uses specifications, i.e., method pre- and post-conditions and class invariants, written in the Java Modelling Language (JML) [21]. The JML tool-set transforms these constructs into run-time assertions that throw JML-specific exceptions when violated. Generating method sequences for methods with JML specifications

amounts to generating *legal* method sequences that satisfy pre-conditions and class invariants, i.e., do not throw exceptions for these constructs. If during the exploration Symstra finds a method sequence that violates a post-condition or invariant, Symstra has discovered a bug; Symstra can be configured to generate such tests and continue or stop test generation. If a class implementation is correct with respect to its specification, paths that throw post-condition or invariant exceptions should be infeasible.

Symstra operates on the bytecode level. It can perform testing of the specifications woven into method bytecode by the JML tool-set or by similar tools. Note that in this setting Symstra essentially uses black-box testing [33] to explore only those symbolic states that are produced by method executions that satisfy pre-conditions and class invariants; conditions that appear in specifications simply propagate into the constraints associated with a symbolic state explored by Symstra. Using symbolic execution, Symstra thus obtains the generation of legal test sequences "for free".

Performance. Based on state subsumption, our current Symstra prototype explores one or more symbolic states that have the isomorphic heap. We plan to evaluate an approach that explores exactly one *union* symbolic state for each isomorphic heap. We can create a union state using a disjunction of the constraints for all symbolic states with the isomorphic heap. Each union state subsumes all the symbolic states with the isomorphic heap, and thus exploring only union states can further reduce the number of explored states without compromising the exhaustiveness of the exploration. (Subsumption is a special case of union; if $C_2 \Rightarrow C_1$, then $C_1 \vee C_2$ simplifies to C_1.)

Symstra enables exploring longer method sequences than the techniques based on concrete arguments. However, users may want to have an exploration of even longer sequences to achieve some test purpose. In such cases, the users can apply several techniques that trade the guarantee of the intra-method path coverage for longer sequences. For example, the user may provide abstraction functions for states [23], as used for instance in the AsmLT generation tool [15], or binary methods for comparing states (e.g. `equals`), as used for instance in Rostra. Symstra can then generate tests that instead of subsumption use these user-provided functions for comparing state. This leads to a potential loss of intra-method path coverage but enables faster, user-controlled exploration. To explore longer sequences, Symstra can also use standard heuristics [33, 9] for selecting only a set of paths instead of exploring all paths.

Limitations. The use of symbolic execution has inherent limitations. For example, it cannot precisely handle array indexes that are symbolic variables. This situation occurs in some classes, such as `DisjSet` and `HashMap` used previously in evaluating Rostra [34]. One solution is to combine symbolic execution with (exhaustive or random) exploration based on concrete arguments: a static analysis would determine which arguments can be symbolically executed, and for the rest, the user would provide a set of concrete values [15].

So far we have discussed only methods that take primitive arguments. We cannot directly transform non-primitive arguments into symbolic variables of primitive type. However, we can use the standard approach for generating non-primitive arguments: generate them also as sequences of method calls that may recursively require more sequences of method calls, but eventually boil down to methods that have only primitive

values (or `null`). (Note that this also handles mutually recursive classes.) JCrasher [13] and Eclat [26] take a similar approach. Another solution is to transform these arguments into reference-type symbolic variables and enhance the symbolic execution to support heap operations on symbolic references. Concrete objects representing these variables can be generated by solving the constraints and setting the instance fields using reflection. However, the collected constraints are often not sufficient to generate legal instances, in which case an additional object invariant is required.

6 Conclusion

We have proposed Symstra, a novel framework that uses symbolic execution to generate a small number of method sequences that reach high branch and intra-method path coverage for complex data structures. Symstra exhaustively explores method sequences with symbolic arguments up to a given length. It prunes the exploration based on state subsumption; this pruning speeds up the exploration, without compromising its exhaustiveness. We have implemented a test-generation tool for Symstra and evaluated it on seven subjects, most of which are complex data structures. The results show that Symstra generates tests faster than the existing test-generation techniques based on exhaustive exploration of sequences with concrete method arguments, and given the same time limit, Symstra can generate tests that achieve better branch coverage than these existing techniques.

Acknowledgments

We thank Corina Pasareanu and Willem Visser for their comments on our implementation of subsumption checking, help in using symbolic execution, and valuable feedback on an earlier version of this paper. We also thank Wolfgang Grieskamp, Sarfraz Khurshid, Viktor Kuncak, and Nikolai Tillmann for useful discussions on this work and anonymous reviewers for the comments on a previous version of this paper. Darko Marinov would like to thank his advisor, Martin Rinard, for supporting part of this work done at MIT. This work was funded in part by NSF grant CCR00-86154. We also acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proc. 6th International Conference on Computer Aided Verification*, pages 484–487, 2004.
2. T. Ball. A theory of predicate-complete test coverage and generation. Technical Report MSR-TR-2004-28, Microsoft Research, Redmond, WA, April 2004.
3. T. Ball and J. R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, 2000.
4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

5. K. Beck. *Extreme programming explained*. Addison-Wesley, 2000.
6. B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
7. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering*, pages 326–335, 2004.
8. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.
9. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.
10. Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. 16th European Conference Object-Oriented Programming*, pages 231–255, June 2002.
11. S. B. Clark W. Barrett. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th International Conference on Computer Aided Verification*, pages 515–518, July 2004.
12. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
13. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
14. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, Palo Alto, CA, 2003.
15. Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.
16. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th SPIN Workshop on Software Model Checking*, pages 235–239, 2003.
18. JUnit, 2003. `http://www.junit.org`.
19. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.
20. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
22. B. Legeard, F. Peureux, and M. Utting. A comparison of the LIFC/B and TTF/Z test-generation methods. In *Proc. 2nd International Z and B Conference*, pages 309–329, January 2002.
23. B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
24. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
25. Microsoft Visual Studio Developer Center, 2004. `http://msdn.microsoft.com/vstudio/`.
26. C. Pacheco and M. D. Ernst. Eclat documents. Online manual, Oct. 2004. `http://people.csail.mit.edu/people/cpacheco/eclat/`.
27. Parasoft. Jtest manuals version 5.1. Online manual, July 2004. `http://www.parasoft.com/`.
28. W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.
29. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. 9th ESEC/FSE*, pages 267–276, 2003.

30. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proc. 2003 Workshop on Software Model Checking*, July 2003.
31. H. Schlenker and G. Ringwelski. POOC: A platform for object-oriented constraint programming. In *Proc. 2002 International Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–170, June 2002.
32. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, 2000.
33. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.
34. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.