Whispec: White-box Testing of Libraries Using Declarative Specifications

Danhua Shao   Sarfraz Khurshid   Dewayne E. Perry

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA
{dshao,khurshid,perry}@ece.utexas.edu

Abstract

We present a novel framework, Whispec, for white-box testing of methods that manipulate structurally complex data, such as those that pervade library classes. Given method preconditions as declarative constraints, our framework systematically generates test inputs for the methods to maximize their code coverage. The constraints are written in Alloy, a first-order language based on relations. To test a method, given its precondition constraint, we first solve that constraint using the Alloy Analyzer and translate a solution into a test input. Next, we execute the method on that input and build the path condition for the resulting execution path. Then, we run the analyzer on a conjunction of the precondition and a new path condition that represents a previously unexplored path. The solution is translated to a new test input, which triggers the next round of test generation. The iterative execution of Whispec can systematically enumerate inputs that maximize code coverage. Experiments using a variety of data structure implementations from the Java libraries show that our framework generates significantly smaller test suites (while maximizing coverage) than those generated by previous specification-based approaches.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *Class invariants*; D.2.5 [**Software Engineering**]: Testing and debugging – *Symbolic execution, Testing tools*; D.3.3 [**Programming Languages**]: Language Contructs and Features – *Constraints*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs – *Assertions, Invariants, Pre- and post-conditions, Specification techniques*.

General Terms Languages, Verification.

Keywords Software Testing, First-order Logic, Symbolic Execution, Contracts.

1. Introduction

Software testing, the most commonly used technique for validating the quality of software, is a labor intensive process, and typically accounts for about half the total cost of software development and maintenance [2]. Automating testing not only reduces the cost of producing software but also increases its reliability.

White-box testing and black-box testing are two commonly used techniques that have complementary strengths. White-box testing uses the internal structures (such as control flow or data flow) of programs. Black-box testing uses an external interface.

Automated approaches to black-box testing make extensive use of specifications, e.g., to specify test inputs or test oracles (correctness criteria). In unit testing of object-oriented code, *preconditions*, which define constraints on legal method inputs, and *postconditions*, which define expected behavior and outputs, form an integral part of the specifications.

TestEra [27] is a specification-based, black-box testing framework for Java programs. Given a method's precondition, TestEra automatically generates all nonisomorphic test inputs up to a given bound, executes the method on each test input, and uses the method's postcondition as a test oracle to check the correctness of each output. A key strength of TestEra is its ability to generate inputs that represent structurally complex data, such as balanced binary search trees, which are often implemented in library classes.

Although TestEra provides efficient enumeration of structurally complex data structures according to given constraints, it does not generate a minimal test suite with respect to some code coverage metrics. A precondition only specifies a method's expected inputs and not the method's implementation details. For example, consider the `contains()` method of Java class `java.util.LinkedList`. Four tests are enough to cover all the branches. However, TestEra generates 120 nonisomorphic tests using a bound of up to 3 elements and 3 non-sentinel nodes.

We propose a novel framework, Whispec, which builds on TestEra and focuses on maximizing code coverage, thereby enabling a specification-based framework for white-box testing. As in TestEra, Whispec requires the user to provide the method's precondition as a declarative constraint. However, in contrast with TestEra, Whispec systematically integrates the precondition with

the control flow of the method and generates test inputs that maximize code coverage.

In Whispec, the specifications are declared as relational first-order logic formulas. As an enabling technology, Whispec uses the Alloy toolset. Alloy [22] is a first-order declarative language based on sets and relations. The Alloy Analyzer [23] is an automatic tool that finds *instances* of Alloy specifications, i.e., finds assignments of values to the sets and relations in the specification such that the specification formulas evaluate to true. Whispec translates Alloy instances into concrete test inputs.

Our key insight is the integration of preconditions and path conditions with relational logic so that they are solved together for test generation. Given the precondition of the method under test, we first solve it with the Alloy Analyzer. The solution is concretized to a test input. Next, we execute the method on that input, observe its execution path, and build path conditions by negating the branch predicates in the execution path. We run the analyzer on a conjunction of the precondition and one of the generated path conditions. The solutions are translated into further test inputs, which exercise previously unexplored paths and trigger the next round of the test generation process. An iterative execution of this approach can systematically enumerate inputs that maximize code coverage.

This paper makes the following contributions:

- **Specification-based white-box approach**. Whispec combines a method's precondition (a declarative formula in first-order logic) with path conditions derived from the method's control-flow, and uses a SAT solver to generate test inputs, which provide a desired level of code coverage.

- **Implementation**. The Whispec prototype provides systematic test generation for Java programs. The prototype uses Breadth First Search to cover different branches as early as possible.

- **Evaluation**. In comparison with TestEra, Whispec generates significantly fewer test cases to achieve the same branch coverage. Each test generated exercises a unique path.

2. Example

This section presents a small illustrative example. Suppose we want to test the `contains()` method of the class `IntList` (which is based on `java.util.LinkedList`) shown in Figure 1.

An object of `IntList` represents a doubly-linked list. The `header` field represents a sentinel node in the list. Objects of the inner class `Entry` represent list nodes. The field `value` represents the (primitive) integer data in a node. The field `next` points to the next node while the field `prev` pointes to the previous node in the list.

The *precondition* for the method `contains()` is given in Alloy as a comment in the source code. This precondition specifies that the structure pointed to by the implicit input parameter, `this`, should be a valid doubly-linked list.

```java
1  public class IntList {
2      private Entry header = new Entry (0, null, null);
3      private int size = 0;
4      private static class Entry {
5          int value;
6          Entry next;
7          Entry prev;
8          Entry(int value, Entry next, Entry prev ){
9              this.value = value;
10             this.next = next;
11             this.prev = prev;
12         }
13     }
14     /* precondition:
15      * // prev is transpose of next
16      * all e1, e2,: this.header.*next |
17      *     e1 = e2.next <=> e2 = e1.prev
18      * //circularity
19      * all e: this.header.*next |
20      *     some e.next && some e.prev
21      * //size OK
22      * this.size = #(this.header.*next - this.header)
23      */
24     public boolean contains(int i) {
25         Entry e = header.next;
26         while ( e != header){
27             if (e.value == i){
28                 return true;
29             }
30             e = e.next;
31         }
32         return false;
33     }
34 }
```

Figure 1. `IntList` example.

The constraint 'all e1, e2: this.header.*next | e1=e2.next <=> e2 = e1.prev' defines the transpose relation between the `next` and `prev` fields. In Alloy, the operator '.' denotes the relational join/composition, '*' denotes reflexive transitive closure, and ':' denotes membership (mathematically, the subset relation). The expression `this.header.*next` thus denotes the set of all nodes reachable from the `header` node of list `this` following 0 or more traversals along the `next` field. The quantifier `all` stands for universal quantification. '<=>' denotes logical equivalence. This constraint specifies that, for every entry `e1` and `e2` in relation 'this.header.*next, if e2.next = e1, e1.prev = e2', and vice versa.

The constraint 'all e : this.header.*next | some e.next && some e.prev' defines the circularity of a list. For each entry `e` in relation 'this.header.*next, e.prev' and 'e.next' should also be an entry within the list. The constraint 'this.size = #(this.header.*next – this.header)' defines the size of a list. The number of entries in the expression 'this.header.*next' minus one should equal the value of 'this.size'.

Given the precondition, we generate tests for `contains()` as follows. First, Whispec solves the *precondition* and translates a solution into a test input. Figure 2 shows an example input, which exercises the path P0: <25, 26, 27, 30, 26, 32> in `contains()`. P0 has three branch statements: 26, 27, and 26. Let *symbols* `IntList_0` and `Int_0` represent the initial values of the input variable `this` and `i`. Then the three *predicates* in path P0 are:
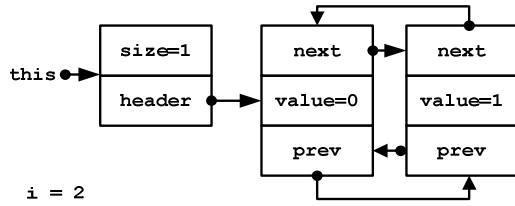
Figure 2. A test generated from the precondition.

Figure 3. A test generated from precondition + path_condition_3.

```
IntList_0.header.next != IntList_0.header
Int_0 == IntList_0.header.next.value
IntList_0.header.next.next != IntList_0.header
```

For Path P0, the decision on the first predicate is true and the decision for each of the other two is false. A conjunction of the predicates or their negations defines a *path condition*—an input that satisfies a path condition exercises that path.

To generate inputs that take different paths, Whispec constructs *new* path conditions by changing the decisions on the predicates. The conjunction of a new path condition and the precondition is passed to a constraint solver, the Alloy Analyzer, and the solution it generates is concretized into a new input.

In this example, we negate each of the three branch decisions in path P0 and construct three new path conditions (Table 1). To illustrate, the conjunction of *path_condition_3* and the precondition generates a new input shown in Figure 3. This input exercises a new path: <25, 26, 27, 30, 26, 27, 30, 26, 32>. Continuing the exploration, Whispec generates four inputs that exercise all four paths in the method `contains()` (using one loop-unrolling).

3. Background

In Whispec, all the constraints, the preconditions and path conditions, are specified in Alloy. Whispec uses the recently developed Kodkod model finder [29] [30] to solve the constraints. In this section, we give a brief overview of Alloy and Kodkod.

3.1 Alloy

Alloy is a first-order declarative language based on sets and relations. The Alloy Analyzer is a tool for automatically analyzing models written in Alloy. The analyzer translates Alloy models into boolean formulas and uses off-the-shelf SAT technology to solve the formulas. The analyzer consists of: a front-end that parses Alloy models into an intermediate representation (IR), a set of optimizations on this IR, and a back-end that translates the IR into boolean formulas.

Table 1. The execution path and the three path conditions for new paths.

Path	predicate_0	predicate_1	predicate_2
Path P0	**True**	False	False
path_condition_1	False		
path_condition_2	True	**True**	
path_condition_3	True	False	**True**
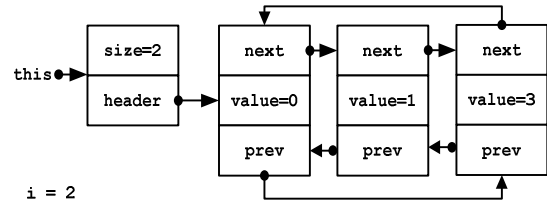
Each Alloy model consists of data (i.e., several sets and relations), several facts (i.e., formulas that put constraints on the data) and an assertion (i.e., a formula to check on the data). These formulas can be structured using functions (i.e., parameterized formulas that can be invoked elsewhere), which the analyzer inlines into the facts and the assertion. Additionally, each analysis specifies a scope (i.e., a bound on the size of basic sets within which to check the formulas). The analyzer translates a conjunction of all facts and the negation of the assertion into a boolean formula such that the boolean formula has a solution if and only if there are some sets and relations that satisfy all the facts and the negation of the assertion (thus providing a counterexample for the assertion).

Alloy is a relational language; every expression in Alloy denotes a relation (or a set in the case of a relation of arity one). Even scalars are represented as singleton sets. More details of the Alloy language are available elsewhere [22].

3.2 SAT

Given a propositional formula over a set of boolean variables, the boolean Satisfiability Problem (SAT) asks whether there exists a variable assignment that makes the formula evaluate to true. SAT is a classical NP-Complete problem; therefore, it is unlikely that there is a polynomial algorithm for solving the SAT problem. However, due to its practical importance in areas such as theorem proving, formal verification, and AI planning, much research effort has been put into developing efficient algorithms for solving SAT problems. Although in the worst case these algorithms require exponential time, in practice current state-of-the-art SAT solvers can often determine the satisfiability of boolean formulas with tens of thousands of variables in a reasonable amount of time [37].

Modern SAT solvers determine the satisfiability of a formula by systematically searching the entire boolean space of the formula. They typically require the input formula to be in the Conjunctive Normal Form (CNF), i.e., a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a positive or negative occurrence of a boolean variable. Some recent SAT solvers can operate without the CNF requirement [15], but the Alloy Analyzer translates all formulas into CNF.

3.3 Kodkod model finder

Kodkod is the latest version of Alloy's translation to SAT. Kodkod provides several new optimizations in the translation, as well as an API to construct Alloy models and to specify the scope, which consists of an upper bound and a lower bound for each relation. The upper bound specifies the tuples that a relation may contain and the lower bound specifies the tuples that the relation must contain. Whispec uses the Kodkod API to run the Alloy Analyzer.

```
1  int abs(int i) {
2      int result;
3      if (i < 0)
4          result = -1 * i;
5      else result = i;
6      return result;
7  }
```

Figure 4. An absolute value calculation program.

3.4 Symbolic Execution

In Whispec, path conditions are generated by symbolic execution. Symbolic execution is a technique for executing a program on symbolic values [25]. There are two fundamental aspects of symbolic execution: (1) defining semantics to operations that are originally defined for concrete values and (2) maintaining a path condition for the current program path being executed—a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider a program (Figure 4) that returns the absolute value of its input.

To symbolically execute this program, we consider its behavior on a primitive integer input, say I. We make no assumptions about the value of I (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them algebraically, e.g., the statement on line 4 updates the value of `result` to be -1 * I. Of course, a tool for symbolic execution needs to modify the type of result to note updates involving symbols and to provide support for manipulating expressions, such as -1 * I.

Symbolic execution of the above program explores the following two paths:

Path 1, [I < 0]: <2, 3, 4, 6>

Path 2, [I >= 0]: <2, 3, 5, 6>

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both. In general, symbolic execution can systematically explore all program paths up to a bounded path length. Tools that implement such exhaustive exploration often use iterated Depth First Search [16][18][28], where the search depth is iteratively increased, until a sufficiency

```
1  void whispec(Method method, Constraint precondition) {
2      LinkedList<PathCondition> pcs = null;
3      pcs.add(new PathCondition(true));
4      while (!pcs.isEmpty()) {
5          PathCondition currentPC = pcs.removeFirst();
6          Solution sol =
7              Alloy.solve(precondition.and(translate(currentPC)));
8          Object input = concretize(sol);
9          pcs.addAll(buildNewPCs(method, input));
10     }
11 }
```

Figure 6. Whispec test generation algorithm.

criterion for exploration has been met. However, in Whispec, we use Breadth First Search in order to cover different branches as early as possible.

4. Whispec Framework

Whispec is a novel framework for automatic test generation for Java programs. We built Whispec on top of a mini Java Virtual Machine, sJVM, to generate path conditions and the Alloy Analyzer to solve the constraints from path conditions and preconditions.

4.1 Framework Outline

Figure 5 illustrates the main components of Whispec and Figure 6 gives an overview of the test generation algorithm. The inputs to the algorithm are the `method` under test and its `precondition`. Before running Whispec to generate tests, we instrument the `method` so that symbolic execution can be performed during the concrete execution.

With the instrumented method and its precondition, Whispec generates tests in the following steps (Figure 6). First, we initialize the `pcs`, a list of path conditions, with only a special boolean formula, `true` (line 3), so that, in the first round execution of the while loop, the constraint solved by `Alloy.solve()` is only the precondition. Then we fetch a path condition from the `pcs` (line 5). With the Alloy Analyzer, we solve the conjunction of the precondition and the fetched path condition (line 6-7). A solution, `sol`, is concretized into a Java object, `input` (line 8). By running the `method` with the `input,` we get the new path conditions and add them into `pcs` (line 9). Next we fetch a path condition from `pcs` and start the next cycle of the test generation process.
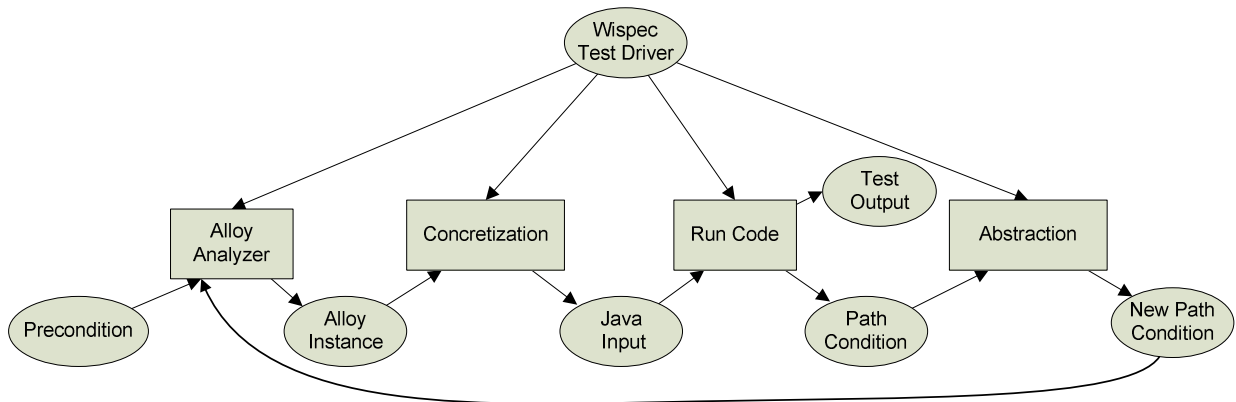

Figure 5. The Whispec Framework

4.2 Path Condition Generation

In the framework of Whispec, path conditions are a critical component. They are generated from the symbolic execution of the current test input and are used to generate new test cases. For a given path, the *path condition* is the conjunction of branch predicates that are required to hold for all the branches along the path.

4.2.1 Instrumentation

In Whispec, path conditions are generated through symbolic execution [25]. Whispec implements symbolic execution through code instrumentation. Before running Whispec, we instrument a method as follows. For each bytecode instruction in the original program, we add a bytecode sequence that performs the same operation on symbolic values.

Table 2 illustrates the spirit of Whispec's instrumentation. For most of the Java bytecode instructions in the original program, Whispec just instruments code to repeat the same operation on sJVM (Symbolic Java Virtual Machine). For example, in the original code, instruction `aload_0` loads the local variable 0 onto the operand stack of JVM. In the instrumented code, we add an instruction, `sJVM.exec_aload(0)`, to load the symbolic value of local variable 0 onto the operand stack of sJVM.

The instrumentation on branch instructions requires additional code. In the last row of Table 2, we add extra code to save the predicate decision from concrete execution into a boolean variable `bc`.

4.2.2 Symbolic Execution on sJVM

sJVM, the execution engine of the instrumented code, is a mini Java Virtual Machine. To accurately emulate the concrete execution, sJVM has a very similar structure to a real JVM.

To effectively represent concrete data values with symbols, we add a *value* field to symbols that representing primitive type values. For a class type value, the symbolic value has a *fields_table* which has the same fields as the class it represents.

To emulate the heap in a real JVM, sJVM uses an array, *symbolic_heap*, to hold all the symbolic values. In sJVM, all symbolic values are referred by their *index* in the *symbolic_heap*. Corresponding to the frame stack in real JVM, *frames* in sJVM has a *local_variable_table* to keep symbolic values for local variables and an *operand_stack* to support evaluations on

Table 2. Bytecode instrumentation.

Before Instrumentation	After Instrumentation
`aload i;`	`aload i;` `sJVM.exec_aload(i);`
`getfield x;`	`getfield x;` `sJVM.exec_getfield(x);`
`iadd;`	`iadd;` `sJVM.exec_iadd();`

`if_acmpeq #;`	`L0: if_acmpeq L1`
	`    bc = false;`
	`    iconst_1;`
	`    goto L2;`
	`L1: bc = true;`
	`    iconst_0;`
	`L2: sJVM.exec_if_acmqeq(bc, L0);`
	`    ifeq #`

symbolic values. All the frames are kept in a *frame_stack*. When a method is invoked, a new stack frame is created and is put on the frame stack. When the callee method returns, the top frame is destroyed.

With the heap and the frame stack, sJVM supports a subset of the instruction set of the real JVM. sJVM can execute most of the load and store instructions, integer arithmetic instructions, object creation and manipulation instructions, operand stack management instructions, control transfer instructions, method invocation and return instructions.

4.2.3 Generation of New Path Conditions

With the support from instrumentation and symbolic execution on sJVM, a path condition will be generated whenever a path is exercised.

Given a path condition, Whispec invokes `generatePathConditions()` (Figure 7) to construct new path conditions for neighboring paths. For example, given a path condition $pc = \{bp_1, bp_2, bp_3, \ldots, bp_n\}$ where bp_i is a branch predicate along the path, Whipsec could generate a set of new path conditions:

$\{negate(bp_1)\}$,

$\{bp_1, negate(bp_2)\}$,

$\{bp_1, bp_2, negate(bp_3)\}$,

...,

$\{bp_1, bp_2, bp_3, \ldots, negate(bp_n)\}$.

In fact, Whispec does not negate all branch predicates, from bp_1 to bp_n. During our iterative test generation process, a predicate is likely to be visited multiple times. Switching back and forth on the same predicate will generate duplicate path conditions. To avoid it, we setup a parameter, `level`, to `generatePathConditions()`. In the path condition pc, only predicate $bp_{level+1}$, $bp_{level+2}$, ..., bp_n are negated to construct new path conditions.

Since all path conditions are kept in a FIFO queue, Whispec uses the BFS (Breadth First Search) strategy to explore the neighboring paths of the current execution path. BFS is efficient in path condition generation. During one round of test generation, Whispec usually generates multiple path conditions. For loop intensive programs, BFS tries to explore new branches as early as possible, rather than going through the same loop condition many times. In Whispec, most of the path conditions are generated from partial paths, not full paths that go from entry to exit. If the path is

```
1  LinkedList<PathCondition>
2  generatePathConditions(LinkedList<BranchPredicate> oldPC, int level)
3  {
4      LinkedList<PathCondition> pcs = new LinkedList<PathCondition>();
5      for (int i = level+1; i < oldPC.size(); i++){
6          PathCondition newPC = new PathCondition();
7          for (int j = 0; j < i; j++){
8              newPC.add(oldPC.get(j));
9          }
10         newPC.add(oldPC.get(i).negate());
11         pcs.add(newPC);
12     }
13     return pcs;
14 }
```

Figure 7. Path condition generation algorithm.

Table 3. Abstraction

Java expression	Alloy expression
Class: `IntList, Entry`	Unary relation: `list, entry`
Variable: `so1, so2`	Singleton unary relation: `r_so1, r_so2`
Field: `next, prev`	Binary relation: `next, prev`
Logic expression: `so1 > so2`	Logic expression: `r_so1.value > r_so2.value`

Table 4. Concretization

Alloy solution	Java data structure

Unary relation:
`entry =`
`  [[a1],[a2],[a3]]`
	Instance:
	`Entry`
	`e1 = new Entry(),`
	`e2 = new Entry(),`
	`e3 = new Entry();`

Binary relation:
`next =`
`  [[a1, a2], [a2, a3]]`
	Field value:
	`e1.next = e2;`
	`e2.next = e3;`

invalid, no effort will be spend on further exploration. This optimizes symbolic execution. To avoid visiting the same path twice, Whispec marks generated path conditions. With a given bound on the size of path condition, Whispec always terminates.

4.3 Abstraction and Concretization

In Whispec, the program under testing is in Java while the constraints are specified in Alloy. To translate between Java expressions and Alloy constraint specifications, we use a relational view of the heap [21]. During abstraction, the path conditions, specified with Java expressions, are translated into Alloy formulas. During concretization, the solution to the constraints will be translated back into concrete Java objects as test inputs.

Table 3 illustrates the abstraction with the `IntList` class defined in Section 2. The translation scheme is type-based. A Java class is mapped to a *unary relation*, in other words, a *set*. The fields of a class are translated to *binary*, *functional relations* that

map objects of this class to the object of the target type. With the mapping from classes and their fields to relations, we can directly translate the logic expressions in predicates into Alloy logical expression.

The solutions to the preconditions and path conditions are specified as a binding of relations to the tuples of a universe of atoms. We also use the `IntList` class defined in Section 2 to illustrate how we concretize the solutions into Java inputs (Table 4). According to the mapping between Java and Alloy in the abstraction phase, we can translate the tuples of atoms back to objects and their fields. This is just the reverse of abstraction. For each relation representing a class, we construct a concrete object of that class for each atom in the tuple set bounded to the relation. For each relation representing the field of a class, we set the values to the corresponding objects' fields. With the concretization of solutions, we construct new test cases to explore unvisited paths.

Table 5. Performance of Whispec in test generation for decision coverage

Data Structure	Method	Scope[1]	Length[2]	Test cases	Explored branches	Decision coverage[3] (%)	#Primary variables (average[5])	#Clauses (average[5])	Time[4] (ms)	Unsatisfied path condition
Linked List	`containts`	3	4	4	3	100	17	83	484	0
	`indexOf`	3	6	6	5	100	10	43	496	0
	`remove`	3	6	6	5	100	10	43	500	0
	`lastIndexOf`	3	6	6	5	100	10	43	500	0
TreeMap	`firstEntry`	3	4	3	2	100	15	198	250	0
	`lastEntry`	3	4	3	2	100	15	198	266	0
	`successor`	6	4	6	5	100	38	856	578	0
	`put`	7	20	53	24	93.8	286	13670	102891	65
Binary Search Tree	`firstEntry`	3	4	3	2	100	13	60	496	0
	`lastEntry`	3	4	3	2	100	13	60	453	0
	`successor`	6	4	6	5	100	27	158	625	0

Scope[1]: the limit on the number of entries in a data structure.

Length[2]: the limit on the number of predicates in a path condition

Decision coverage[3]: the coverage for both the true and false decisions for all the branches, including the functions directly or indirectly called by the function under test.

Time[4]: total time for test generation and test execution.

Average[5]: the average of # variables or # clauses over all test cases.

Figure:

10000

1000

100

120 120 120 120 201 4011 53 11 11 3133

10

10 10 6 6 6 6 6 4

1

LinkedList.contains() LinkedList.IndexOf() LinkedList.Remove() LinkedList.LastIndexOf() TreeMapFirstEntry() TreeMap.LastEntry() TreeMap.successor() TreeMap.put() BinarySearchTree.FirstEntry() BinarySearchTree.LastEntry() BinarySearchTree.successor()
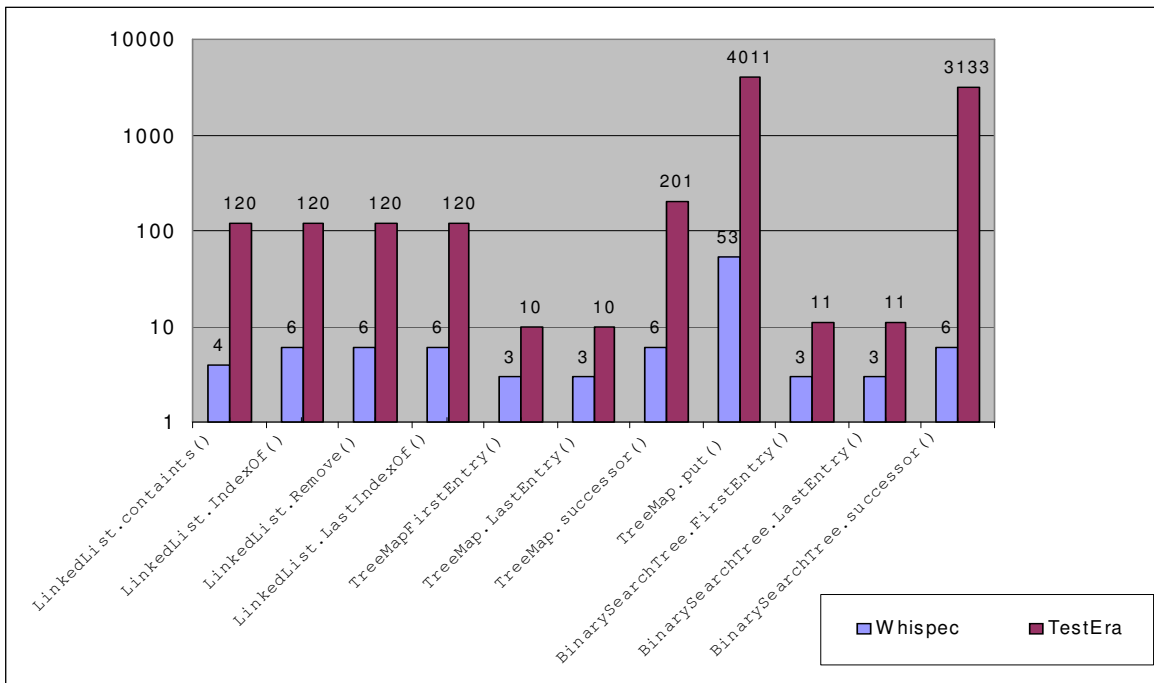
□ Whispec   ■ TestEra

Figure 8. Comparison between Whispec and TestEra in test generation with the same decision coverage

5. Experiments

To evaluate the effectiveness of Whispec, we run tests on some methods of Java classes: LinkedList, TreeMap, and BinarySearchTree. LinkedList implements doubly-linked circular lists in the Java Collections Framework. Each list has a size field and a header node, which is a sentinel node. TreeMap implements the Map interface using red-black trees. This implementation uses binary trees with parent pointers. Each node (implemented with inner class Entry) has a key and value. BinarySearchTree is an implementation of binary search tree. Each node has an integer value. A tree satisfies binary search constraints: left subtree of a node contains only values less than the node's value, and the right subtree of a node contains only values greater than or equal to the node's value.

Given a limit on the number of entries in a data structure and the number of predicates in a path condition, we run Whispec to generate tests to cover all the branch decisions. We use Kodkod API to specify the constraints and select miniSAT [12] as the SAT solver. We performed the experiments on an Intel Pentium D processor with 1GB RAM.

Table 5 shows the experimental results. In the setting of the experiment, the scope is the limit on the entries in a data structure. The length is the limit on the predicates in a path condition. The time taken includes both the test generation and test execution because these two parts are mixed together in Whispec. To maximize coverage, we try to cover all the branches with both true and false decisions. For the methods invoked inside the methods under test, we also explore the branches inside the callee methods.

For most of the methods, Whispec can generate test cases covering all the branch decisions within 2 minutes. The exception is the put() method of TreeMap where 3 of the 48 branch decisions are not covered. After manually checking the uncovered branch decisions, we found that all the 3 branch decisions are unsatisfiable and the paths that go through them are infeasible.

To evaluate the efficiency of Whispec, we also compared the number of test generated by Whispec with the number that TestEra (or Korat) generates. TestEra needs to construct all the non-isomorphic data structures for every input parameter (include the implicit parameter this), and combine them using a cross-product to produce test cases. The results in Figure 8 show that, with the same branch decision coverage, Whispec generates significantly fewer test cases than TestEra. For example, to cover the 45 valid branch decisions in the put() function of TreeMap, Whispec generates 53 test cases while TestEra generates 4011 test cases.

6. Limitations

This section addresses some limitations of Whispec.

Primitive types, such as integer, are very commonly used in preconditions and implementations. However, Alloy Analyzer, the constraints solver of Whispec, does not efficiently support primitive type arithmetic operations, such as integer addition. A promising approach is proposed in [11] to handle larger integers with relational logic. The use of this technique could extend the application of Whispec.

We envision enabling the Whispec framework to use (in conjunction with SAT solvers) specialized decision procedures for handling operations on a variety of primitive types. One way to do so is to partition the constraints that describe an input according to whether they constrain a primitive type or a reference type, and

use a SAT solver to solve the structural constraints and use appropriate decision procedures for solving constraints on primitive data.

Our current implementation does not support checking the exceptional behavior of programs or generating inputs with multi-dimensional-array-based components.

Inheritance is fundamental in object-oriented programming. So far, we have not addressed how to utilize class hierarchies in test generation.

Multi-threaded programming is common in software development. Dynamically checking the correctness of multi-threaded programs for deadlocks and race conditions requires the ability to control thread scheduling. We envision using a model checker in conjunction with a SAT solver to check such programs.

7. Related Works

As a hybrid approach, Whispec has similarities with both black-box testing and white-box testing approaches.

In black-box testing, the closest approaches are TestEra [27] and Korat [4]. In TestEra, structural constraints are specified with Alloy and solved with SAT solvers. TestEra prunes isomorphic solutions to effectively reduce redundant test cases. Korat uses constraints specified in Java. Korat performs a systematic search of the space of possible structures and uses the given constraints as a basis for efficiently pruning its search and generating valid structures. QuickCheck, a product of QuviQ AB, is also a specification-based test generation tool. QuickCheck is developed from Claessen and Hughes' earlier QuickCheck tool for Haskell [7] and a re-design for Erlang [1]. QuviQ QuickCheck uses high-order functional languages, such as Haskell or Erlang, to specify the properties, such as pre-/post-conditions, a program should satisfy. According to the property specification, QuickCheck randomly generate test inputs. Different from Whispec, none of them considers the constraints from path conditions and therefore they are not efficient at generating tests that focus on maximizing coverage.

In white-box testing, symbolic execution and branch negation are in common use for test generation [25]. DART [18] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. However, DART focuses only on path conditions involving integers. To overcome the path explosion in large programs, SMART [17] introduced inter-procedural static analysis techniques to reduce the paths to be explored by DART. CUTE [28] extends DART to handle constraints on references. CUTE can in principle be used with preconditions on structural inputs. In [26], random searching is integrated with CUTE to increase the ability to get deep paths. Although Whispec also uses symbolic execution and negates branch conditions, it differs from approaches based on DART and CUTE in constraint solving and new path exploration. Whispec uses first-order logic formulas to specify the constraints whereas DART and CUTE require imperative assertions as specifications. Whispec conjoins the precondition and path condition and solves them together whereas CUTE handles them separately, which forces it to explore a much larger number of paths since it must explore paths within the precondition as well as those within the method body; DART does not handle preconditions. In addition, Whispec uses BFS,

rather than the DFS (Depth First Search) that is used in DART and CUTE, to cover different branches as early as possible. This approach is particularly effective in generating tests for branch coverage. For loop intensive programs, DART and CUTE will go inside loop and repeat the same branch condition until reaching the bound. Whispec prefers the path going out of the loop first and can reach new branches earlier.

Agitator [3], a commercial tool that can automate test generation, performs static and dynamic analysis on Java code and generates tests according to the identified behaviors. In Agitator, the constraints are provided by Daikon[13]'s dynamic invariant detection algorithm and path conditions are solved by data type specific constraint solvers, such as string and integer solvers. In contrast with Whispec, Agitator does not focus on data structure constraints.

EGT [6] and EXE [5] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of the symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. However, neither handles preconditions or class invariants.

A combination of model checking and symbolic execution is proposed to do white-box testing in [24] [31]. To solve the constraints from path conditions and preconditions, they do symbolic execution on both code in the function under test and code for path conditions. While this approach provides a tight integration of preconditions and path conditions, it requires preconditions to be given in Java and its efficiency depends heavily on how the user writes the preconditions.

Csallner's DSD [9] uses the invariant inference tool Daikon [13] to get class invariants, and ESC/Java [14] to do static checking, CnC [8] to solve constraints, and JCrasher [10] to generate and run tests with. Although both static analysis and dynamic execution are used, they work separately in the whole process. The symbolic execution is not integrated with concrete execution, and the effectiveness in pruning false positives also depends on the invariant inference tool.

Symstra [34] generates unit tests for complex data structures through symbolic state exploration and symbolic execution. Its performance is much better than its predecessor Rostra [33], which is a concrete execution approach. However, Symstra requires a complete implementation and does not consider preconditions.

There are various other approaches for generating test cases based on program paths. In [20], tests are generated by iteratively switching branch conditions to cover different branches. In [32], the shape of an input data structure is constructed by solving alias between object references. In [35][36], they do symbolic execution and analyze structure pointers and array index expressions to generate test cases going through designated paths. In [20], test cases are generated by constructing the control flow graph, searching paths to specified points and solving the constraints for the selected path. However, none of them propose the integration of preconditions with test generation.

8. Conclusions

We presented a novel framework, Whispec, for white-box testing of methods that manipulate structurally complex data, such as those that pervade library classes. Given method preconditions as

declarative constraints, our framework systematically generates test inputs for the methods to maximize their code coverage. The constraints are written in Alloy, a first-order language based on relations. To test a method, given its precondition constraint, we first solve that constraint using the Alloy Analyzer and translate a solution into a test input. Next, we execute the method on that input and build the path condition for the resulting execution path. Then, we run the analyzer on a conjunction of the precondition and a new path condition that represents a previously unexplored path. The solution is translated to a new test input, which triggers the next round of test generation. The iterative execution of Whispec can systematically enumerate inputs that maximize code coverage. Experiments using a variety of data structure implementations from the Java libraries showed that our framework generates significantly smaller test suites (while maximizing coverage) than those generated by previous specification-based approaches.

9. Acknowledgments

10. References

[1] T. Arts and J. Hughes. Erlang/quickcheck. In Ninth International Erlang/OTP User Conference, Nov. 2003.

[2] B. Beizer. Software Testing Techniques. International Thomson Computer Press, 1990.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In Proc. ACM/SIGSOFT Int. Symp. Software Testing and Analysis, pages 169–180. Portland, Maine, USA, 2006.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, July 2002.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In Proc. of the 13th ACM Conference on Computer and Communications Security (CCS), Alexandria, Virginia, October-November 2006.

[6] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In Proc. of the 12th International SPIN Workshop on Model Checking of Software, San Francisco, CA, August 2005

[7] K. Claessen and J. Hughes. Quickcheck: a lightweight tool forrandom testing of haskell programs. In ICFP, pages 268–279, 2000.

[8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, MO, May 2005.

[9] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In Proc. of 2006 International Symposium on Software Testing and Analysis (ISSTA), Portland, Maine, July 2006.

[10] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. Software: Practice and Experience, 34:1025–1050, 2004.

[11] J. Dolby, M. Vaziri, and F. Tip. Finding Bugs Efficiently with a SAT Solver. In Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), Dubrovnik, Croatia, September 2007.

[12] N. Ee′n, and N. So¨rensson. An extensible SAT-solver. Theory and Applications of Satisfiability Testing (SAT 2003), Vol 2919:5 of LNCS, Springer, 2004.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27(2):99–123, Feb. 2001.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), June 2002.

[15] M. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In Proc. 39thConference on Design Automation (DAC), Jun. 2002.

[16] P. Godefroid. Model checking for programming languages using VeriSoft. In Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL), pages 174–186, Paris, France, January 1997.

[17] P. Godefroid. Compositional dynamic test generation. In Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), Nice, France, January 2007.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI), Chicago, IL, June 2005.

[19] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In Proc. of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Clearwater Beach, Florida, March 1998.

[20] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In Proc. of the 15th International Conference on Automated Software Engineering (ASE), Grenoble, France, September 2000.

[21] D. Jackson. Object Models as Heap Invirants. In *Essays on Programming Methodology,* edited by Annabelle McIver and Carroll Morgan. Springer Verlag, 2000.

[22] D. Jackson. Software Abstractions: logic, language, and analysis. MIT Press, Cambridge, MA, 2006.

[23] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In Proc. 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 2000.

[24] S. Khurshid, C. Pasareanu and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. The 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Warsaw, Poland. Apr 2003.

[25] J. C. King. Symbolic execution and program testing. Communications of the ACM, Volume 19, Issue 7, July 1976.

[26] R. Majumdar and K. Sen. Hybrid Concolic Testing. In Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, MN, USA, May 2007.

[27] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In Proc. of the 16th IEEE International Conference on Automated Software Engineering (ASE), San Diego, CA, November 2001.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE), Lisbon, Portugal September, 2005.

[29] E. Torlak and G. Dennis. Kodkod for Alloy Users. First ACM Alloy Workshop, Portland, Oregon, November 2006.

[30] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. The 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Braga, Portugal, March 2007.

[31] W. Visser, C. S. Pasareanu, S. Khurshid. Test Input Generation with Java PathFinder. In Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Boston, MA. July 2004.

[32] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In Proc. of the 17th IEEE International Conference on Automated Software Engineering (ASE), September 2002.

[33] 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 (ASE), September 2004.

[34] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Proc. of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2005.

[35] Z. Xu and J. Zhang. A Test Data Generation Tool for Unit Testing of C Programs. In Proc. of the 6th International Conference on Quality Software (QSIC), 2006.

[36] J. Zhang. Symbolic execution of program paths involving pointers and structure variables. In Proc. of the 4th International Conference on Quality Software (QSIC), 2004

[37] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In Proc. 8th Conference on Automated Deduction (CADE), July 2002.