Test Input Generation with Java PathFinder

Willem Visser
RIACS/USRA
NASA Ames Research Center
Moffett Field, CA 94035, USA
wvisser@email.arc.nasa.gov

Corina S. Păsăreanu
Kestrel Technology
NASA Ames Research Center
Moffett Field, CA 94035, USA
pcorina@email.arc.nasa.gov

Sarfraz Khurshid
UT ARISE
University of Texas at Austin
Austin, Texas 78712, USA
khurshid@ece.utexas.edu

ABSTRACT

We show how model checking and symbolic execution can be used to generate test inputs to achieve structural coverage of code that manipulates complex data structures. We focus on obtaining branch-coverage during unit testing of some of the core methods of the red-black tree implementation in the Java `TreeMap` library, using the Java PathFinder model checker. Three different test generation techniques will be introduced and compared, namely, straight model checking of the code, model checking used in a black-box fashion to generate all inputs up to a fixed size, and lastly, model checking used during white-box test input generation. The main contribution of this work is to show how efficient white-box test input generation can be done for code manipulating complex data, taking into account complex method preconditions.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Testing and Debugging—Symbolic Execution

General Terms: Algorithms, Verification

Keywords: Testing Object-oriented Programs, Model Checking, Symbolic Execution, Coverage, Red-Black Trees

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 [9]. Automating testing would not only reduce the cost of producing software but also increase the reliability of modern software. A recent report by the National Institute of Standards and Technology estimates that software failures currently cost the US economy about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost [1].

Automated test case generation has been well studied in the literature (see Section 6), but most of this work has focused on the generation of test inputs containing simple (unstructured) data. In this paper we'll address the problem of doing test input generation for code that manipulates complex data structures. The main research challenge in this area is how to do efficient test input generation that will obtain high code coverage — we will show how symbolic execution over complex data can address this problem.

Model checking [13] has been hugely popular for the last two decades. More recently the application of model checking to the analysis of software programs has also come to the fore [7,15,27,43]. Model checking programs however is hard due to the complexity of the code and it often cannot completely analyze the program's state space since it runs out of memory. For this reason some of the most popular program model checkers rely on (predicate) abstractions [7,27] to reduce the size of the state space, but these techniques are not well suited for handling code that manipulates complex data — they introduce too many predicates, making the abstraction process inefficient. We will show that although a program model checker (without relying on abstraction) cannot always achieve good code coverage when dealing with programs manipulating complex data, augmenting it with symbolic execution (which can be seen as a form of abstraction), can result in the generation of tests that will achieve high code coverage.

There has been an active research community investigating the generation of test inputs with the use of model checking [3,4,21,26,28] — the focus is on specification-based test input generation (i.e. black-box testing) where coverage of the specification is the goal. Model checking lends itself to test input generation, since one simply specifies as a set of (temporal) properties that a specific coverage cannot be achieved and the model checker will find counterexamples, if they exist, that then can easily be transformed into test inputs to achieve the stated coverage goal.

Symbolic execution has long been advocated as a means for doing efficient test input generation [31], but most of the ensuing research has focused on generating tests for simple data types (integers for the most part).

In previous work [30] we developed a verification framework based on symbolic execution and model checking that handles dynamically allocated structures (e.g. lists and trees), simple (primitive) data (e.g. integers and strings) and concurrency. The framework uses method preconditions to initialize fields only with valid values and method postconditions as test oracles to test a method's correctness.

We show here how we used and extended the symbolic execution framework from [30] to perform automated test input generation for unit testing of Java programs. To gen-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-820-2/04/0007 ...$5.00.

erate inputs that meet a given testing criterion for a particular method under test, we model check the method. The testing criterion is encoded as a set of properties the model checker should check for. Counterexamples to the properties represent paths that satisfy the coverage criterion. Symbolic execution (which is performed during model checking) computes a representation (a set of constraints) of all the inputs that execute those paths. The actual testing requires solving the input constraints in order to instantiate test inputs that can then be executed.

The framework uses *lazy initialization*, i.e. it initializes the components of the method inputs on an "as needed" basis, without requiring an a priori bound on input sizes. While [30] describes in detail how symbolic execution and lazy initialization are used during model checking, we highlight here the use of complex preconditions during lazy initialization, to initialize the inputs only with valid values.

In particular, we highlight a key feature of our framework: the use of preconditions that are *conservative*, i.e. they may be evaluated on partially initialized structures and return `false` only if the initialized fields of the input structure violate a constraint in the precondition. This important feature was mentioned briefly in [30]; we elaborate on it here.

We show how lazy initialization in combination with the use of conservative preconditions during initialization to eliminate incorrect structures results in a powerful and efficient way of performing symbolic execution of code that manipulates complex structured data. We also show here how we solve the input constraints in order to get the test inputs that are necessary for the actual testing.

To illustrate the flexibility of our framework, we contrast the above white-box technique with a black-box technique where we use the method preconditions to systematically generate all the (non-isomorphic) test inputs up to a given size; this is done by symbolically executing the code for the precondition. This latter approach has similarities to the techniques employed in the Korat tool [10] which also executes the code for the precondition, but it does not use symbolic execution (for the primitive data) and lazy initialization.

We evaluate our approaches by generating tests for the red-black tree [16] implementation in the Java `TreeMap` library. The contributions of our work are:

- A powerful and flexible test input generation framework for unit testing. The framework uses an efficient approach to the symbolic execution of code manipulating complex data structures, that takes into account preconditions to stop the analysis of infeasible paths as soon as possible. The framework can be used uniformly both for white-box and black-box testing.

- We show how our framework can be used for generating tests for code manipulating complex data structures - specifically red-black trees.

- We illustrate the flexibility of model checking as a tool for test input creation by comparing straight model checking of the code under test, a black-box approach and a white-box approach.

2. BACKGROUND

We describe here the Java PathFinder (JPF) model checker [43] that has been extended with a symbolic execution capability. We show in Section 4 how we use this extension of JPF for white-box and black-box test input generation.

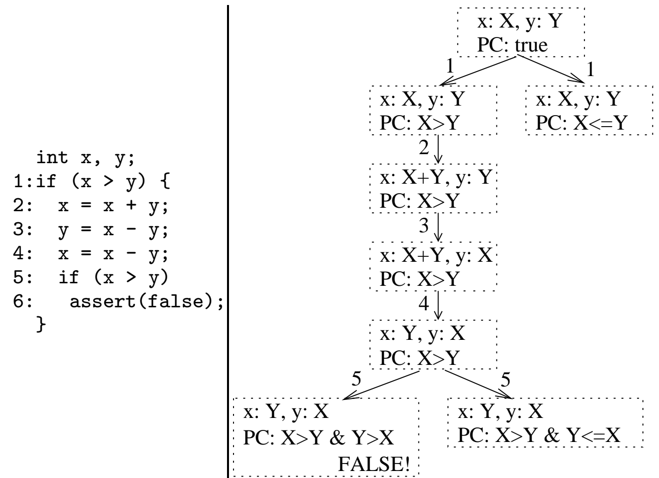

Figure 1: Code that swaps two integers and the corresponding symbolic execution tree, where transitions are labelled with program control points

2.1 Java PathFinder

JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). JPF can handle all of the language features of Java and it also treats nondeterministic choice expressed in annotations of the program being analyzed — annotations are added to the programs through method calls to a special class `Verify`. The following methods from the `Verify` class will be used in this paper:

`randomBool()` returns a boolean value nondeterministically.

`random(n)` returns values $[0, n]$ nondeterministically.

`ignoreIf(cond)` forces the model checker to backtrack when `cond` evaluates to true.

JPF has previously been used to find errors in a number of complex systems including the real-time operating system DEOS from Honeywell [39] and a prototype Mars Rover developed at NASA Ames (called K9) [11]. More recently it was also used as a means for generating input plans that the current K9 rover takes as input [5] — the plans were generated in a black-box fashion similar to the technique described in section 4.2.

2.2 Symbolic Execution in Java PathFinder

In this section we give some background on symbolic execution and we present the symbolic execution framework used for reasoning about Java programs.

2.2.1 Background: Symbolic Execution

The main idea behind symbolic execution [31] is to use *symbolic values*, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`)

and a program counter. The path condition is a (quantifier free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment in Figure 1, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`. Figure 1 also shows the corresponding symbolic execution tree. Initially, `PC` is *true* and `x` and `y` have symbolic values `X` and `Y`, respectively. At each branch point, `PC` is updated with assumptions about the inputs, in order to choose between alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and `PC` is updated accordingly. If the path condition becomes `false`, i.e. there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement `(6)` is unreachable.

2.2.2 Generalized Symbolic Execution

In [30] we describe an algorithm for generalizing traditional symbolic execution to support advanced constructs of modern programming languages, such as Java and C++. The algorithm handles dynamically allocated structures, primitive data and concurrency. We have since extended the work in [30] by adding support for symbolic execution of arrays.

The algorithm starts execution of a method on inputs with *uninitialized* fields and uses *lazy initialization* to assign values to these fields, i.e. it initializes fields when they are first accessed during the method's symbolic execution. This allows symbolic execution of methods without requiring an a priori bound on the number of input objects.

When the execution accesses an uninitialized reference field, the algorithm nondeterministically initializes the field to `null`, to a reference to a new object with uninitialized fields, or to a reference of an object created during a prior field initialization; this systematically treats aliasing. When the execution accesses an uninitialized primitive (or string) field, the algorithm first initializes the field to a new symbolic value of the appropriate type and then the execution proceeds.

When a branching condition on primitive fields is evaluated, the algorithm nondeterministically adds the condition or its negation to the corresponding path condition and checks the path condition's satisfiability using a decision procedure. If the path condition becomes infeasible, the current execution terminates (i.e. the algorithm backtracks).

2.2.3 Framework

Our symbolic execution framework is built on top of the JPF model checker. To enable JPF to perform symbolic execution (and lazy initialization), the original program is instrumented by doing a source to source translation that adds nondeterminism and support for manipulating formulas that represent path conditions[1].

[1]The interested reader is referred to [30] for a detailed description of the code instrumentation

```
  class Node {
    int elem;
    Node next;
  }
  /* precondition: acyclic() */
  void foo() { ...
1:  if (elem > t.elem)
2:    next = t.next;
  }
}
```

Figure 2: Simple example to illustrate generalized symbolic execution

The model checker checks the instrumented program using its usual state space exploration techniques — essentially, the model checker explores the symbolic execution tree of the program. A *state* includes a heap configuration, a path condition on primitive fields, and thread scheduling. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure, such as the Omega library [40] for linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks.

The framework can be used for test input generation and for finding counterexamples to safety properties. For test input generation, the model checker generates paths that are witnesses to a testing criterion encoded as a set of properties. For every reported path, the model checker also reports the input heap configuration (encoding constraints on reference fields), the path condition for the primitive input fields, and thread scheduling, which can be used to reproduce the error.

Note that performing (forward) symbolic execution on programs with loops can explore infinite execution trees. This is why, for systematic state space exploration, the framework uses depth first search with iterative deepening or breadth first search.

Although we concentrate in this paper on the analysis of sequential code, it is worth mentioning that our symbolic execution framework handles concurrency, as it uses the model checker to systematically analyze thread interleavings. Using a model checker as a search engine for our framework allows us to also exploit other built-in capabilities of the model checker, such as backtracking, different search capabilities (e.g. heuristic search), and techniques that combat state-explosion (e.g. partial order and symmetry reductions). We should also note that, although we consider here branch coverage as a metric for testing, our framework can handle other testing criteria that can be encoded as properties the model checker should check for (e.g. data-flow based coverage).

2.2.4 Illustration

We illustrate the generalized symbolic execution on a simple example (see Figure 2). Class `Node` implements singly-linked lists; the fields `elem` and `next` represent, respectively, the node's integer value and a reference to the next node.

Figure 3 gives (part of) the corresponding code that was instrumented for symbolic execution: concrete types were replaced with "symbolic types" (library classes that we provide) and concrete operations with method calls that implement "equivalent" operations on symbolic types. Class `Expression` supports manipulation of symbolic integers.

```
class Node {
  Expression elem;
  Node next;
  boolean _next_is_initialized = false;
  boolean _elem_is_initialized = false;

  static Vector v = new Vector();
  static {v.add(null);}

  Node _new_Node() {
    int i = Verify.random(v.size());
    if(i<v.size()) return (Node)v.elementAt(i);
    Node n = new Node();
    v.add(n);
    return n;
  }
  Node _get_next() {
    if(!_next_is_initialized) {
      _next_is_initialized=true;
      next = Node._new_Node();
      Verify.ignoreIf(!precondition());//e.g. acyclic
    }
    return next;
  }

  void foo() { ...
1:  if(_get_elem()._GT(t._get_elem()))
2:     _set_next(t._get_next());
  }
}
```

Figure 3: Instrumented code

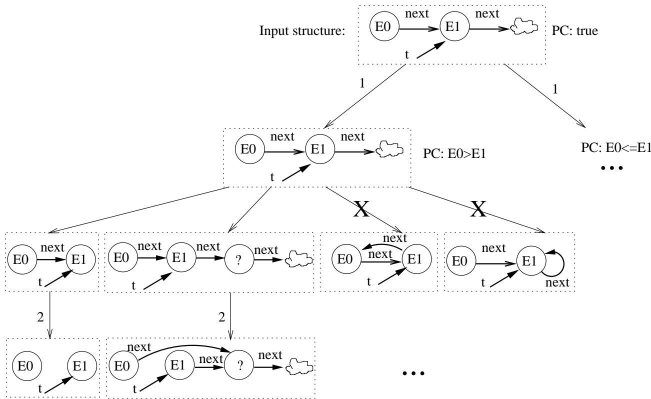

Figure 4: Symbolic execution tree (excerpts)

Field reads and updates are replaced by **get** and **set** methods (**get** methods implement the lazy initialization). For each field in the original class declaration, boolean fields (i.e. `_next_is_initialized` and `_elem_is_initialized`) are added. These fields are set to true by **get** (**set**) methods. Vector `v` stores the input objects that are created as a result of lazy initialization. The helper method `_new_Node`, which is called by `_get_node`, uses the elements in vector `v` to systematically initialize input reference fields, according to different aliasing possibilities.

Figure 4 illustrates the paths that are generated during the symbolic execution of the code of method `foo`, for a given input structure. Each node of the execution tree denotes a symbolic state. Branching in the tree corresponds to a nondeterministic choice that is introduced to build a path condition or to handle aliasing. Edges labelled with numbers refer to program statements and those without numbers to lazy initialization steps. The value "?" for an `elem` field indicates that the field is not initialized and the "cloud" indicates that the `next` field is not initialized; `null` nodes are not represented. Structures represent constraints on reference fields, e.g. the input structure in Figure 4 represents all (cyclic or acyclic) lists with at least two nodes such that `t` points to the second node.

As we will explain later in Section 4.3, method preconditions can be used during lazy initialization: if the input structure violates the precondition, the model checker backtracks (i.e. call to method `Verify.ignoreIf` in Figure 3). For example, if we consider the precondition that the input list should be acyclic, the algorithm does not explore the transitions marked with an "X" in Figure 4.

3. CASE STUDY: RED-BLACK TREES

We have used our input generation techniques for testing a variety of programs, including methods of classes in the `java.util` package. Most of these programs manipulate complex data structures. In particular, we illustrate our techniques on the Java implementation of red-black trees given in `java.util.TreeMap` from the standard Java libraries (version 1.3). Red-black trees [16] are binary search trees with one extra bit of information per node: its *color*, which can be either red or black. By restricting the way nodes are colored on a path from the root to a leaf, red-black trees ensure that the tree is balanced, i.e. guarantee that basic dynamic set operations on a red-black tree take $O(log\ n)$ time in the worst case.

A binary search tree is a red-black tree if:

1. Every node is either red or black.

2. The root is black.

3. If a node is red, then both its children are black.

4. Every simple path from the root node to a descendant leaf contains the same number of black nodes.

All four of these *red-black properties* are expressible in Java. We use these four properties together with the basic properties of binary search trees to define a `repOk` method, i.e. a Java predicate that checks the representation invariant (or class invariant) of the corresponding data structure. In this case, `repOk` checks if the input is a red-black tree. Figure 5 gives part of the `java.util.TreeMap` declaration and Figure 6 gives a fragment of code from `repOk` representing the third red-black tree property: red nodes have only black children. The generation of red-black trees is particularly interesting for our framework, due to their complex structure (i.e. they have primitive fields, back pointers, etc.).

In the next section, we present different techniques that we used to generate test inputs for the implementation of red-black trees in `java.util.TreeMap`. We will illustrate test input generation for several helper methods for the `put` and `remove` methods in class `java.util.TreeMap`, which are responsible for adding and deleting the node corresponding

```
public class TreeMap {
  Entry root;
  static final boolean RED = false;
  static final boolean BLACK = true; ...
  static class Entry implements Map.Entry {
    Object key;
    Object value;
    Entry left;
    Entry right;
    Entry parent;
    boolean color; ...
  }

  /* precondition: repOk(root); */
  public Object remove(Object key) {...}

  public Object add(Object key) {...}
  ...
}
```

Figure 5: Declaration of `java.util.TreeMap`

```
boolean repOk(Entry e) {
  ...
  // RedHasOnlyBlackChildren
  workList = new LinkedList();
  workList.add(e);
  while (!workList.isEmpty()) {
    Entry current=(Entry)workList.removeFirst();
    Entry cl = current.left;
    Entry cr = current.right;
    if(current.color == RED) {
      if(cl != null && cl.color == RED) {
        return false;
      }
      if(cr != null && cr.color == RED) {
        return false;
      }
    }
    if (cl != null)
      workList.add(cl);
    if (cr != null)
      workList.add(cr);
  }
  ...
  return true;
}
```

Figure 6: Method `repOk` (excerpts).

to a given key from the tree. We should note that deletion is the most complex operation among the standard operations on red-black trees and involves rotations. Together with the auxiliary methods, addition together with deletion in `java.util.TreeMap` are about 300 lines of Java code. The (implicit) precondition for both `put` and `remove` methods requires the input to satisfy its class invariant (i.e. `repOk`): the input must be a red-black tree.

4. TEST INPUT GENERATION

In this section we will illustrate three applications of model checking to the test input generation of software manipulating a complex data structure. We will focus on the implementation of the `put` and `remove` methods for red-black trees in the Java `TreeMap` library. As a testing criterion we use source code level branch-coverage since we want to compare the black-box to the white-box approaches to test input generation. As mentioned in Section 2.2.3 our framework can also handle other kinds of testing criteria.

We aim to generate a set of *non-isomorphic* tests, which meets the desired coverage criteria. Isomorphism among tests is defined as isomorphism among graphs where the heap of a Java program is viewed as an edge-labeled graph: node identities are permutable, while primitive values are not [10]. Note that the Java semantics do not allow object allocation to dictate the exact object identities, which implies that initializing a test input (at the concrete representation level by setting field values or at the abstract level using a sequence of method invocations) more than once (say for regression testing) does not generate identical structures but it generates isomorphic structures.

First we show how a model checker can be used to do the testing, by executing sequences of method calls in the data structure's interface. Secondly, we show how we can use our symbolic execution framework to build all (non-isomorphic) input trees up to a given small size that are to be used for the (black-box) testing of the method. This is done by symbolically executing the Java code of the method's precondition (in this case the code of `repOk`).

Lastly, we show how our framework can be used for white-box test input generation and how conservative preconditions are used during lazy initialization, to stop the analysis of infeasible paths. We also show how the input constraints computed by symbolic execution are solved to provide the inputs for the actual testing.

4.1 Model Checking as Testing

When doing model checking there is a clear distinction between the system being analyzed and the environment of the system, i.e. the inputs that the system takes. Whenever the environment is under-approximated (less behaviors are considered than are present in the actual environment) during model checking then model checking becomes a form of testing. Note that more often than not this is the case during the model checking of software, since the environment is usually very large. Considering this connection, one can therefore use a model checker to generate inputs and analyze the code on those inputs.

To illustrate this idea we show how one can test the Java `TreeMap` library by analyzing all sequences of `put` and `remove` calls on a set with maximally N elements using the JPF model checker (Figure 7). Note that, for this example, we are more interested in the coverage of the code, rather than correctness and hence we only use the model checker's default properties (uncaught exceptions being the most important here) as an oracle — in general our approach allows more general oracles, including method postconditions expressed as Java predicates.

4.2 Input Generation For Black-box Testing

Our framework can be used to automatically generate Java data structures from a description of method precon-

```
public static int N = 5;
public static TreeMap t = new TreeMap();
public static Integer[] elems;
static {elems = new Integer[N];
  for (int i = 0;i< N; i++)
    elems[i] = new Integer(i);
}
public static void main(String[] args) {
  while (true) {
    if (Verify.randomBool())
      t.put(elems[Verify.random(N-1)],null);
    else
      t.remove(elems[Verify.random(N-1)]);
} }
```

Figure 7: Model checking as testing

ditions. Note that for sequential code, generalized symbolic execution explores only paths on non-isomorphic inputs. Therefore, we can generate non-isomorphic input structures that satisfy a precondition, by applying generalized symbolic execution to the code of the precondition. Once we have an input structure, we use off-the-shelf constraint solvers for solving the constraints in the path condition, thus obtaining the test input. Test inputs can then be used for black-box testing of the method under test. The drawback of this approach is that there is no relationship between the inputs and the code coverage. On the other hand, if one would be interested in covering the input specification rather than the code under test, this black-box method achieves full coverage of the input structures up to a given bound.

In order to test the `put` and `remove` methods we automatically generated all (non-isomorphic) input trees up to a given small size from the Java description of the method's precondition (i.e. the structural invariant), thus eliminating the need to construct the inputs using a sequence of method calls. Our framework symbolically executes `repOk` and it generates the input structures whenever `repOk` returns true. We put a limit on the number of generated objects: whenever the size of the vector that stores the objects created during lazy initialization (see Section 2) reaches that limit, the model checker backtracks. As a result, all the input structures satisfying `repOk` with size up to the specified limit are created.

Our approach can be contrasted with a *brute force* approach, where one will first generate all possible trees up to a given size according to the class definition, and then would apply `repOk` to select only valid red-black trees. Our approach scales better since we generate trees on demand (with lazy initialization) and we backtrack as soon as a red-black tree property is violated, thus pruning large portions of the search space. It is important to note that the actual structure of `repOk` is crucial to the efficiency of our method. If `repOk` would first evaluate the tree and only at the end determine whether the tree is valid, our approach would be equivalent to generating all trees before pruning.

4.3 Input Generation for White-box Testing

Our symbolic execution framework can be used for input generation during white box testing. To generate inputs that meet a given testing criterion, our framework is used to symbolically execute the method under test and to model check it against properties that encode the testing criterion. Counterexamples to the properties represent paths that satisfy the criterion. For every path, our framework also reports an input structure and a path condition on the primitive input values, which together define a set of constraints that the inputs should satisfy in order to execute that path[2].

A particular characteristic of our framework is that it uses method preconditions during two phases of the input generation to eliminate infeasible structures:

- a *conservative* precondition, that can deal with partially initialized structures, is used during lazy initialization (see Section 4.3.1)
- when a counterexample is found the structural constraint for the path is used as input to a *concrete* precondition (the same one used in Section 4.2) to solve the constraints with only valid inputs (see Section 4.3.3)

4.3.1 Conservative Preconditions

We use preconditions in initializing fields (see Figure 3). In particular, a field is not initialized to a value that violates the precondition. Notice that we evaluate a precondition on a structure that still may have some uninitialized fields, therefore we require the precondition to be *conservative*, i.e. return `false` only if the *initialized* fields of the structure violate a constraint in the precondition.

Consider the analysis of the `remove` method in the red-black tree implementations. The method has as precondition the class invariant of the red-black tree data structure (i.e. the `repOk` Java predicate). The conservative version of the precondition that we used during lazy initialization is illustrated in Figure 8. Boolean flags `_left_is_initialized` and `_right_is_initialized` were added by our code instrumentation to keep track of uninitialized input fields (see Section 2.2.4). The code of `conservative_repOk` is identical to that of `repOk`, with the exception that the constraints encoded in `repOk` are only evaluated on initialized fields.

As an example, assume that there are three input trees as illustrated in Figure 9 that are created during the analysis of the `remove` method (at a lazy initialization step). Round filled nodes represent entries colored black and empty nodes represent entries colored red; null nodes are not represented. As before, a "cloud" denotes an uninitialized field — intuitively representing a set of nodes, since it can be lazily initialized to different nodes. For simplicity of presentation, we omit to represent the `key` and `value` fields. Figure 9 also shows the results of evaluating the `conservative_repOk` on the tree structures. The first tree violates the `repOk`, no matter what the concrete value of the "cloud" is, since red nodes cannot have red children. In this case the model checker will backtrack and it will not consider this structure any further. The second tree is a concrete structure that satisfies `repOk`. The third tree represents concrete trees that may or may not satisfy `repOk`; `conservative_repOk` returns `true` (or rather Don't Know), and the analysis continues.

We should note that the lazy initialization of input fields in our framework is related to *materialization* of summary nodes in shape analysis [33], while the conservative preconditions can be formulated in the context of abstract interpretation. We would like to explore these connections further.

[2]Our framework also reports the thread scheduling information, in the case of multi-threaded code

```
boolean conservative_repOk(Entry e) {
  ...
  // RedHasOnlyBlackChildren
  workList = new LinkedList();
  workList.add(e);
  while (!workList.isEmpty()) {
    Entry current=(Entry)workList.removeFirst();
    Entry cl = current.left;
    Entry cr = current.right;
    if (current.color == RED) {
      if (current._left_is_initialized &&
          cl != null && cl.color == RED) {
        return false;
      }
      if (current._right_is_initialized &&
          cr != null && cr.color == RED) {
        return false;
      }
    }
    if (current._left_is_initialized &&
        cl != null)
      workList.add(cl);
    if (current._right_is_initialized &&
        cr != null)
      workList.add(cr);
  }
  ...
  return true;
}
```

Figure 8: Predicate conservative_repOk

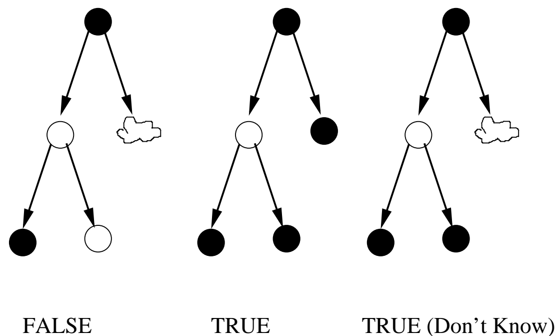

Figure 9: Evaluation of conservative_repOk on 3 structures

4.3.2 Handling Destructive Updates

The lazy initialization algorithm builds the input structures on an as needed basis, when they are first accessed during symbolic execution. If the code under analysis performs destructive updates, the structure of the inputs can be lost. To create test inputs we therefore need to reconstruct these input structures. As an example, consider the structures in the leaves of the symbolic execution tree depicted in Figure 4, which are the result of a destructive update; these structures no longer contain the information that in the input structure, there is a link between the first two nodes. In order to recover the original input structures, we keep mappings between objects with uninitialized fields and objects that are created when those fields are initialized; these mappings are used to reconstruct input structures. Note that whenever the precondition needs to be evaluated during lazy initialization, we first reconstruct the input structures and then we evaluate the precondition on these reconstructed input structures.

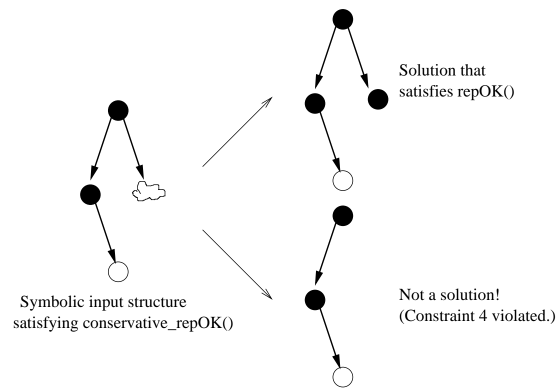

Figure 10: Concretization of symbolic structures

4.3.3 Solving Constraints

The result of symbolically executing a particular program path is a heap structure, that encodes constraints on reference fields, and a path condition, that encodes constraints on primitive data. These constraints define the inputs that will allow the execution of the path. In order to obtain the actual test inputs, we have to solve these constraints, i.e. we have to build actual Java data structures that can be used during testing. In our framework, we solve these constraints separately for structures and primitive data.

We first "convert" symbolic structures to concrete structures that have no uninitialized fields and that also satisfy the method's precondition. If there are no method preconditions, this amounts to simply setting all the uninitialized fields to `null`. Otherwise, the symbolic input structure (which satisfies the conservative precondition) is used as input to the code for the concrete precondition which is symbolically executed to obtain the concrete structure, in a way similar to the test input generation method described in Section 4.2. The constraints in the path condition are then solved using an off-the-shelf constraint solver.

We should note that not all the concretizations of a symbolic structure that satisfies the conservative precondition, are valid solutions. An example is given in Figure 10.

5. EXPERIMENTS

As mentioned before we measured branch-coverage for the methods of the `TreeMap` class and for simplicity we will only consider here 3 helper methods: `deleteEntry` (del), `fixAfterDeletetion` (fixD) and `fixAfterInsertion` (fixI). The optimal[3] branch-coverage that can be achieved for these methods are 86% (19 out of 22) for `deleteEntry`, 100% (20 out of 20) for `fixAfterDeletion` and 88% (14 out of 16) for `fixAfterInsertion`. Note that 100% statement coverage can be obtained for all 3 methods - the uncovered branches are all for *missing* `else` branches where the `if` option is always taken.

[3]The rest of the branches can be shown to be infeasible.

	Resources			% Coverage		
N	Time	Mem	States	del	fixD	fixI
1	1.7	2.2	28	18	0	0
2	1.9	2.2	104	68	5	6
3	2.4	2.3	730	68	5	75
4	7.8	3	10109	86	90	88
5	127	8.8	194525	86	90	88
6	Out of Memory					

Table 1: Model Checking as Testing

	Statistics				% Coverage		
N	Structs	Candidates	Time	Tests	del	fixD	fixI
1	1(1)	5	2.4	2	18	0	6
2	3(2)	24	2.9	8	68	5	6
3	5(2)	103	4.7	16	72	50	88
4	9(4)	432	12	36	86	90	88
5	17(8)	1830	44	84	86	100	88
6	33(16)	7942	212	196	86	100	88

Table 2: Black-box Structural Tests

Statistics			% Coverage		
Time	Mem	Tests	del	fixD	fixI
72	5	11 (53)	86	100	88

Table 3: White-box Tests

All results were obtained using the JPF model checker (version 3.1.1) on a 2.2 Ghz dual processor Pentium with 1 Gb of memory running Windows 2000 with Java 1.4.2.

5.1 Model Checking as Testing

The results in Table 1 show the coverage achieved in the three methods when model checking the code in Figure 7 for different values of N (the number of entries to be added to the `TreeMap`). The coverage numbers were gathered using JPF's facility to calculate branch-coverage on the bytecode level during execution — a simple transformation was done to obtain branch-coverage on the source code level. Note that this technique is also used for the branch-coverage calculations in the rest of the section. The table also contains the time taken (in seconds), the memory consumed (in Mb) and the number of states generated during model checking.

The results indicate that this approach does not scale well. Until size 4 the results give the appearance of actually being very good (fast, low memory and reasonable coverage), but the exponential explosion becomes apparent at size 5 and size 6 cannot be handled in its entirety. For `fixAfterDeletion`, 100% branch-coverage is not achieved; `fixAfterDeletion` is called from `deleteEntry` hence it is not too surprising that there is a large jump in its coverage from size 3 to 4.

An advantage of the model checking approach is that it obtains good path (behavioral) coverage for small input domains as well as for systems where testing is fundamentally hard, such as concurrent systems. In the domain considered here, namely (sequential) programs manipulating complex data, this technique of testing could be an appropriate first pass at finding errors, but to obtain good structural coverage one needs a more sophisticated approach.

5.2 Black-box

Table 2 shows the results for black-box structural coverage up to a fixed size (N). The input trees were generated by doing a symbolic execution of the `repOk` method using JPF. For each of the input trees all possible node deletions and one new insertion were then executed (Tests) and the code coverage measured. We also report the total number of trees created (Structs) up to a specific size N (in parenthesis only the number of trees of size N is given) as well as how many structures where considered (Candidates) by the lazy initialization of the code within `repOk`. Since the memory consumption was minimal (less than $10Mb$ for the cases shown) we only report on the time taken to generate the trees — the time for running the tests were negligible. Note that since the structures are generated up to a given size, the results include all the smaller structures as well.

After 1 minute all the trees required to achieve optimal coverage of the code were generated — size 5. Note that the trees of size N created by analyzing all sequences of `put` and `remove` operations (Section 5.1) are a subset of the trees of size N allowed by `repOk` — we believe this is due to the class invariant (`repOk`) being more permissive than the property maintained by sequences of `put`/`remove` operations. Indeed (say for performance concerns) methods may maintain properties that are stronger than the stated class invariant, thereby disallowing certain structures that are otherwise valid from being generated during executions of method sequences.

5.3 White-box

The results from doing a white-box analysis of the methods of `TreeMap` to obtain a set of test inputs to achieve optimal branch-coverage of the three methods is given in Table 3. We report on the time taken in seconds, memory usage in Mb, the number of tests run (with the number of tests generated before removing duplicates in parenthesis) and the coverage obtained. Although we only mention the input trees we are considering, each test input consists of an input tree as well as the node to put/remove.

Note that here we don't parameterize the results with the size of the trees (as in Table 2), since the goal is to cover all branches and that is achieved with different size trees. We do however limit the size of the trees that we are looking for to size 5 and smaller. As to be expected the coverage obtained is optimal. There are many duplicates amongst the input trees generated to cover all the 53 branches in the code — only 11 unique input trees are required. The 11 trees are made up of all trees of size 1, 2, 3 and 4, but only two trees of size 5 (out of the 8 possible trees).

5.4 Discussion

The lazy initialization of `repOk` that we use for black-box test input generation can be compared to the approach taken by Korat [10]. Korat generates inputs from constraints given as Java predicates and it uses backtracking (as we also do) but it monitors executions of `repOk` on fully initialized inputs within an a priori given input size. For the example used here the number of candidate structures the two techniques consider is very similar. However, if one also introduces integer values in the structures, our approach considers a lot fewer structures, since it uses symbolic execution together

with integer constraint solving whereas Korat has to enumerate the integer fields.

The fact that in the white-box approach only 11 test inputs are required versus the 84 to obtain the same (optimal) coverage in the black-box approach illustrates the power of using a goal-directed white-box approach over a *blind* black-box approach to test input generation for obtaining high coverage. We believe that for more complicated structural invariants the difference in test input size for the black and white box approach would be even more pronounced.

A drawback of our current approach is that we cannot determine whether code is unreachable when the code contains cycles — as is the case in the red-black tree examples considered here. We are considering techniques such as automatic invariant generation [38] and the use of shape-predicates and abstraction to address this problem. An inherent drawback of symbolic execution is the strength of the decision procedures used to check for infeasible path conditions. Currently we can only deal with linear integer constraints, but we hope to add more powerful decision procedures in the near future.

We should note that we have experimented with different approaches to representing data structures, e.g. they could be completely symbolic or represented as partially initialized Java structures, as in the context of the work presented here. We used this latter approach because it facilitates the evaluation of preconditions written as Java predicates.

6. RELATED WORK

6.1 Specification-based Testing

The idea of using constraints to represent inputs dates back at least three decades [14, 29, 31, 41]; the idea has been implemented in various tools including EFFIGY [31], TEGTGEN [32], and INKA [24]. But most of the prior work has been to solve constraints on primitive data, such as integers and booleans, and not to solve constraints on complex structures, which requires very different constraint solving techniques.

Some recent frameworks, most notably TestEra [35] and Korat [10,34], do support non-isomorphic generation of complex structures, such as red-black trees. TestEra generates inputs from constraints given in Alloy, a first-order declarative language based on relations. TestEra uses off-the-shelf SAT solvers to solve constraints. We have already discussed about Korat in Section 5.4. The Korat algorithm has recently been included in the AsmL Test Generator [20] to enable generation of structures. TestEra and Korat focus on solving structural constraints. They do not directly solve constraints on primitive data and instead, systematically try all primitive values within given bounds, which may be inefficient. Further, TestEra and Korat have been used for black-box testing and not in a white-box setting.

An early paper by Goodenough and Gerhart [23] emphasizes the importance of specification-based testing. Various projects automate test case generation from specifications, such as Z specifications [18], UML statecharts [37], ADL specifications [12], or AsmL specifications [25]. These specifications typically do not involve structurally complex inputs and they do not address object-oriented programs.

Doong and Frankl [19] use algebraic specifications to generate tests (including oracles) for object oriented programs. Their ASTOOT tool generates sequences of interface events and checks whether the resulting objects are observationally equivalent (as specified by the algebraic specification). Although here we were only interested in generating tests covering the input structures (black-box) and code (white-box), using an algebraic specification to create additional tests and check the functional requirements of the code is a straight-forward extension.

Gargantini and Heitmeyer [21] use a model checker to generate tests that violate known properties of a specification given in the SCR notation. Ammann and Black [3, 4] combine model checking and mutation analysis to generate test cases from a specification. Rayadurgam *et al.* use a structural coverage based approach to generate test cases from specifications given in RSML^{-e} by using a model checker [26]. Lee *et al.* [28] define a framework for using temporal logic to specify data-flow test coverage.

6.2 Static Analysis

The Three-Valued-Logic Analyzer (TVLA) [33, 42] is the first static analysis system that can verify preservation of the list structure in programs that perform list reversals via destructive updates to the input list. TVLA has been used to analyze small programs that manipulate doubly linked lists and circular lists, as well as some sorting programs. Recently, the TVLA system was extended to also deal with preconditions on shape-graphs [44]. The pointer assertion logic engine (PALE) [36] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers. These data structures include doubly linked lists, trees with parent pointers, and threaded trees.

While static analysis of program properties is a promising approach for ensuring program correctness in the long run, the current static analysis techniques can only verify limited program properties. For example, none of the above techniques can verify correctness of implementations of balanced trees, such as red-black trees. Testing, on the other hand, is very general and can verify any decidable program property for realistically large implementations, but for bounded inputs.

6.3 Software Model Checking

There has been a lot of recent interest in applying model checking to software [7,8,15,17,22,27,43]. Most of this work has focused on checking event sequences, specified in temporal logic or as “API usage rules” in the form of finite state machines. These approaches offer strong guarantees: if a program is successfully checked, there is no input/execution that would lead to an error. However, they typically did not consider linked data structures or considered them only to reduce the state space to be explored and not to check the data structures themselves.

Our work shows how to enable an off-the-shelf model checker to check for properties of complex structures, taking into account complex preconditions. Our algorithms can be implemented in a straightforward fashion to enable other model checkers that support dynamic structures to check structural properties too.

Recently two popular software model checkers BLAST and SLAM, both based on predicate abstraction, were used to do white-box test input generation [2, 6]. In both cases the goal is to generate tests that will cover a specific predicate or a combination of predicates. These techniques do

not focus on generating complex test inputs and they can not handle complex preconditions.

7. CONCLUSION

The main contribution of this work was to show how complex preconditions can be used to allow efficient symbolic execution of code manipulating complex data to generate test inputs obtaining high code coverage. In particular we illustrated how to use a conservative precondition that evaluates symbolic structures (i.e. structures that are not fully initialized) to eliminate structures that cannot lead to valid concrete inputs that will achieve the stated coverage. This conservative precondition can be seen as an abstract version of the concrete precondition, since it will disallow invalid structures, but might accept structures that can be instantiated to concrete structures that will fail the concrete precondition. Although we created the conservative precondition by hand, we would like to investigate how to generate it directly from the concrete precondition by using techniques from abstract interpretation. In our experiments the precondition was the class invariant, but the approach can handle any precondition expressed as a Java predicate.

We also showed two other approaches to using model checking and symbolic execution for testing. Firstly, the most traditional approach from a model checking perspective, where one simply applies model checking to the system under test. This approach can obtain high levels of behavioral coverage, but only for small configurations of data structures. Secondly, we showed that by symbolically executing the code for the precondition one can efficiently obtain tests suitable for black-box testing. This second approach shows the flexibility of our lazy initialization approach to symbolic execution, and it resembles the algorithm employed by the Korat tool [10] that has been highly successful in generating test inputs.

We believe that a flexible approach to testing complex software is very important. To this end we think the techniques covered here can be seen as a continuum in the following fashion. If the code to be analyzed doesn't have a full specification, one can use the black-box approach that only considers the structure of the inputs to generate tests. Note that the structure of the inputs must be known and in our case given as a Java predicate. If a specification does exist (for example an algebraic specification as used in [19]) then a specification centered approach to test input generation can be used to augment the above. Note, this requires the specification to be given in a notation acceptable to a model checker, again Java in our case. At this point the black-box approach has been exhausted and one needs to consider the code (white-box) to generate additional tests. Although we only considered simple coverage criteria here to drive the symbolic execution based test input generation, one can specify any criteria that can be expressed as properties to the model checker, for example, predicate coverage [2,6]. Lastly, testing is not sufficient in all cases, for example testing a concurrent program is notoriously incomplete, and a more powerful technique such as program model checking can then be used.

Acknowledgments

We would like to thank Darko Marinov for insightful discussions and for sharing with us his Java implementation of the class invariant for red-black trees. We would also like to thank the reviewers for their detailed comments that allowed us to greatly improve the paper.

8. REFERENCES

[1] The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Planning report 02-3, May 2002.

[2] R. J. Adam J. Chlipala, Thomas A. Henzinger and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.

[3] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the 4th IEEE International Symposium on High Assurance Systems and Engineering*, 1999.

[4] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, 1998.

[5] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the 10th International Workshop on Abstract State Machines*, Taormina, Italy, March 2003.

[6] T. Ball. Abstraction-guided test generation: A case study, 2003. Microsoft Research Technical Report MSR-TR-2003-86.

[7] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.

[8] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 1–3, 2002.

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

[10] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[11] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. In *Proceedings of the SEI/CM Software Model Checking Workshop*, Pittsburgh, March 2003. To Appear in Formal Methods in System Design Journal.

[12] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proc. 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 285–302, Sept. 1999.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, MA, 1999.

[14] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Sept. 1976.

[15] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[17] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, July 1999.

[18] M. R. Donat. Automating formal specification based testing. In *Proc. Conference on Theory and Practice of Software Development*, volume 1214, pages 833–847, Lille, France, 1997.

[19] R.-K. Doong and P. G. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, 1994.

[20] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[21] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Springer-Verlag, 1999.

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

[23] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[24] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, FL, 1998.

[25] 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 (ISSTA)*, pages 112–122, July 2002.

[26] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, Oct. 2003.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International SPIN Workshop on Model Checking of Software*, volume 2648 of *LNCS*, 2003.

[28] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, April 2002.

[29] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3), 1975.

[30] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[32] B. Korel. Automated test data generation for programs with procedures. San Diego, CA, 1996.

[33] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symposium*, Santa Barbara, CA, June 2000.

[34] D. Marinov. *Testing Using a Solver for Imperative Constraints*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004. (to appear).

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

[36] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.

[37] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, Oct. 1999.

[38] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*. Springer-Verlag, 2004.

[39] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, Limeric, Ireland., June 2000. ACM Press.

[40] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), Aug. 1992.

[41] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[42] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, Jan. 1998.

[43] W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model Checking Programs . *Automated Software Engineering Journal*, 10(2), April 2003.

[44] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, April 2004.