An Evaluation of Exhaustive Testing for Data Structures

Darko Marinov Alexandr Andoni Dumitru Daniliuc Sarfraz Khurshid Martin Rinard
MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139
{marinov,andoni,dumi,khurshid,rinard}@csail.mit.edu

ABSTRACT

We present an evaluation of exhaustive testing of linked data structures with sophisticated structural constraints. Specifically, we use the Korat testing framework to systematically enumerate all legal inputs within a certain size. We then evaluate the quality of this test suite according to several measurements: ability to detect injected faults in the original correct implementations, code coverage, and specification coverage. Our results indicate that it is feasible to use exhaustive testing to obtain, within a reasonable amount of time, a high-quality test suite that can detect almost all faults and achieve complete code and specification coverage. Moreover, our results show that our exhaustive tests are of higher quality than randomly selected test suites that contain the same number of inputs selected from a larger potential input set. We conclude that exhaustive testing is a practical and effective testing methodology for sophisticated linked data structures.

1. INTRODUCTION

Testing is currently the dominant method for finding and eliminating software errors and, as such, is critical to the ability of the software industry to produce high-quality code. Obtaining good test cases is obviously a key requirement to successfully testing any software artifact, but many issues complicate this activity. Requiring developers to manually provide test cases is labor intensive (especially for linked data structure with complex structural properties) and may produce test cases that exercise only a restricted subset of the functionality of the software.

The alternative is to automatically generate a range of test cases, then filter out any test cases that do not satisfy the required input invariants of the system under test. While this approach may work well for systems with simple input invariants, it may be prohibitively expensive for systems with complex input invariants—the density of test cases that satisfy the invariant in the search space may be so small that the generator is unable to produce legal inputs within a reasonable amount of time.

We have developed Korat [6], a technique for generating test cases that satisfy complex invariants. Korat enables the developer to provide an operational way of identifying the legal inputs. Specifically, the developer provides a precondition predicate, written in a standard programming language, that returns true if the input satisfies the required invariant and false otherwise. Korat processes this predicate to efficiently produce a stream of structures that satisfy the property identified by the pre-condition. To check the correctness of the implementation, Korat tests the implementation on the generated inputs to verify that the execution satisfies the provided post-condition. Given a bound on the size of inputs, called the *scope*, Korat generates all inputs within the scope that satisfy the invariant. We call testing with such inputs *exhaustive testing*.

1.1 Evaluation

In theory, exhaustive testing could detect any error in the implementation. In practice, time constraints make it possible to test the implementation only up to certain scope, raising the possibility that the resulting incomplete test suite may fail to detect an error. To evaluate the effectiveness of exhaustive testing, we have used it to test a benchmark set of standard data structure implementations. Our evaluation centers around two issues: the quality of the test suite and the performance of the test case generation algorithms.

We use *mutation testing* [15, 24, 41] to measure the quality of the test suites that Korat generates. Mutation testing first produces a set of new (potentially) faulty versions of a program, called *mutants*, by performing syntactic modifications on the program. It then measures how many mutants a test suite detects. A test suite is *mutation-adequate* if it detects a desired percentage of mutants. Results show that a test suite that detects a high percentage of injected faults is likely to detect real faults [41].

We evaluate the following hypotheses for exhaustive testing of our data structure benchmarks:

- **Mutation:** There is a certain small scope that satisfies mutation-adequacy criterion for data structures.
- **Coverage:** Mutation-adequacy criterion is stronger than complete code coverage criterion.
- **Feasibility:** Korat can generate inputs and check correctness for the mutation-adequate scope.
- **Randomness:** Exhaustive test suites are of higher quality than randomly selected test suites that contain the same number of inputs selected from a larger input set.

1.2 Correctness

The correctness of our evaluation depends on the correctness of Korat. The correctness of Korat is also important in practice—if Korat mistakenly produces a structure that does not satisfy the input invariant, the developer may waste valuable time attempting to track down a non-existent error in the implementation. If, on the other hand, Korat incorrectly causes a set of structures to be systematically omitted, the generated test cases will miss any error that is triggered only by omitted structures.

We address these concerns by providing 1) a formalization of Korat's test case generation algorithm, and 2) a proof that this algorithm is both sound (it only generates structures that satisfy the input invariants) and complete (it generates all such structures). This proof provides a strong guarantee that should increase the confidence of the developer in the correctness of the test case generation tool.

1.3 New Technique

We present a new technique, *input property exploitation*, that increases the effectiveness of exhaustive test case generation. This technique uses *dedicated* generators to optimize the generation of common properties that often appear within input invariants. It enables the test case generator to substantially prune the search without eliminating any structures that satisfy the input invariant. We evaluate the effectiveness of exploiting common input invariant properties by comparing the performance of our Korat-based implementation with and without this technique. Our results show that the use of this technique can speed up the performance of test case generation for up to 75%. Moreover, dedicated generators make it easier to write pre- and post-conditions.

1.4 Contributions

This paper makes the following contributions:

- **Evaluation:** It presents an evaluation of exhaustive testing for data structures. Our results show that:
  - Exhaustive testing within small scope can generate mutation-adequate test suites.
  - Exhaustive test suites can achieve complete coverage for even smaller scopes, but such suites do not detect all mutants.
  - It is feasible to use Korat to generate inputs and check correctness for these scopes.
  - Exhaustive testing within some scope is often more effective than random testing with somewhat bigger inputs.

  We anticipate that our results will extend to a wide range of programs whose inputs must satisfy complex structural invariants.
- **Correctness:** It formalizes the test case generation algorithm and presents a proof of its correctness.
- **Input Property Exploitation:** It shows how to exploit the presence of common input properties to prune the search during test case generation.
- **Ferastrau:** We present design and implementation of a tool for mutation testing of Java programs.
- **Novel Applications:** We illustrate how to apply Korat to white-box testing and to testing sequences of method calls.

```
class SearchTree {
    Node root; // root node
    int size;  // number of nodes in the tree
    static class Node {
        Node left;  // left child
        Node right; // right child
        Comparable info; // data
    }

    /*@ normal_behavior // non-exceptional specification
      @     // precondition
      @     requires repOk();
      @     // postcondition
      @     ensures repOk() && !contains(info) &&
      @             \result == \old(contains(info));
      @*/
    boolean remove(Comparable info) { ... }

    boolean repOk() {
        // checks that empty tree has size zero
        if (root == null) return size == 0;
        // checks that the input is a tree
        if (!isAcyclic()) return false;
        // checks that size is consistent
        if (numNodes(root) != size) return false;
        // checks that data is ordered
        if (!isOrdered(root)) return false;
        return true;
    }
}
```

Figure 1: Example code and specification.

2. EXAMPLE

This section illustrates how programmers can use Korat to test their programs. As a running example, we use a method for removing an element from a set implemented as a binary search tree. Even though removal is conceptually simple, the implementation involves intricate details to restore a tree after removing an inner node. Figure 1 shows Java code that declares a binary tree and its `remove` method. Each object of the class `SearchTree` represents a binary search tree. The `size` field contains the number of nodes in the tree. Objects of the inner class `Node` represent nodes of the trees. The elements of the set are stored in the `info` fields. The elements implement the interface `Comparable`, which provides the method `compareTo` for comparisons. Appendix A shows the full code for the example `remove` method.

The example `remove` method is annotated using the Java Modeling Language (JML) [33]. The `normal_behavior` keyword specifies that if the precondition (keyword `requires`) is satisfied before the method, then the method must satisfy the postcondition (keyword `ensures`) at the end and must return without raising an exception. The method `repOk` is a Java predicate that checks the *representation invariant* [35] of the corresponding data structure. For illustrative purposes, we put `repOk` in the pre/post-conditions; in practice, it is usually given as a class invariant (keyword `invariant`) that is implicitly conjoined with the pre/post-conditions [33]. Good programming practice suggests that implementations of abstract data types provide these predicates, as they are useful for checking correctness of the implementations [35].

In this example, `repOk` checks if the input is a valid binary search tree with the correct `size` First, `repOk` checks if the tree is empty. If not, `repOk` checks that there are no undirected cycles along `left` and `right`, that the number of nodes reachable from `root` is `size`, and that all elements in the left (right) subtree of a node are smaller (larger) than

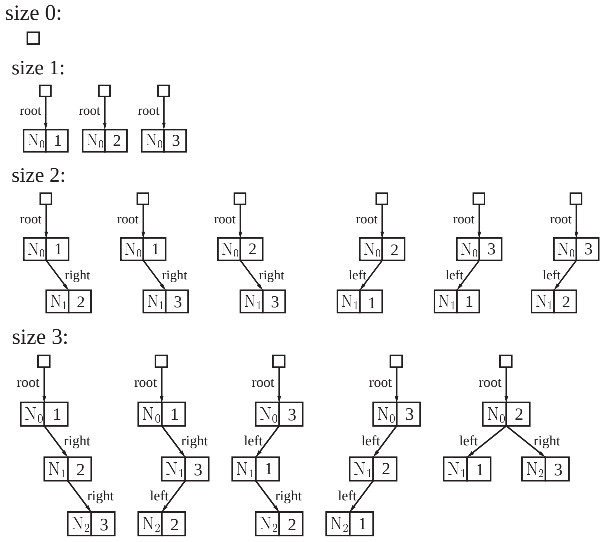

Figure 2: Trees generated for scope three.

the element in that node. Appendix A shows the full code for `repOk` (and the methods it invokes). The same `repOk` is also used for `add` and other methods in `SearchTree`. Manually developing a high-quality test suite for all methods in a data structure is typically much harder than writing a `repOk` invariant that Korat uses to automatically generate test inputs. The method `contains` checks that the tree contains the given element. The JML keyword `\result` denotes the return value of the method; `remove` returns `true` iff it removes an element from the tree. The JML keyword `\old` denotes that its expression should be evaluated in the pre-state, i.e., the state immediately before the method's invocation.

To test the `remove` method in a black-box setting, we use Korat to generate valid inputs for the method. Each input is a pair of a tree and an element. The precondition defines valid inputs: the tree satisfies `repOk`, and the element is unconstrained. To limit the number of inputs, Korat uses a *finitization* (Section 3.2) that specifies bounds on the number of objects to be used to construct data structures and the values stored in the fields of these objects. For trees, finitization gives the maximum number of nodes and the possible elements; a tree is in scope s if it has at most s nodes and s elements. Two trees are *isomorphic* if they have the same branching structure and isomorphic elements, irrespective of the identity of the actual nodes or elements in the trees.

Given a finitization and scope, Korat generates all non-isomorphic input pairs that satisfy the precondition. For example, in scope three, Korat generates 45 input pairs that are the Cartesian product of the 15 trees (shown in Figure 2) and the three elements. For `SearchTree`, we use Korat to generate inputs and check correctness of `remove` and `add` methods. In scope seven, Korat generates 41300 input pairs for both these methods in less than ten seconds. With dedicated generators (Section 5), it takes less than three seconds to generate these inputs.

Korat uses the JML tool-set [8, 33] to translate method postconditions (and JML assertions) into Java runtime assertions. After generating the inputs, Korat invokes the method, with assertions, on each input and reports a counterexample if the method fails to satisfy the postcondition. This process checks the correctness of the method for the given scope. For example, for scope seven, it takes less than two seconds to check both `remove` and `add` for all 41300 inputs.

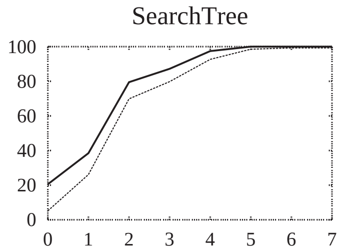

Figure 3: Variation of statement coverage (thick line) and rate of mutant killing (thin line) with scope.

Figure 3 shows how coverage and the rate of mutant killing vary with the scope for the `SearchTree` benchmark. For `remove`, the mutation testing compares the outputs that consist of both the `boolean` return value and the value of the receiver tree in the post-state, i.e., the state immediately after the method's invocation. Scope five is sufficient to achieve complete coverage, but scope six is required to kill all non-equivalent mutants. Generating inputs and checking correctness for these scopes takes less than 15 seconds.

3. KORAT

This section describes how Korat finds inputs that satisfy a Java predicate. We first give an informal overview of Korat. We then describe parts of Korat most relevant for formalization and extensions; more details on other parts can be found in [6]. For illustration, we use the `repOk` method from `SearchTree` as the predicate, and we show how Korat generates valid trees.

3.1 Overview

Given a Java predicate and a bound on its input, Korat generates all non-isomorphic inputs that are *valid*, i.e., inputs for which the predicate always returns `true`. Korat uses a *finitization* (Section 3.2) to bound the *state space* (Section 3.3) of predicate inputs. Korat uses backtracking (Section 3.4) to systematically explore this state space. Korat generates *candidate inputs* and invokes the predicate on them to check their validity.

Naive checking of all possible candidate inputs would prohibit searching very large state spaces. Korat uses two optimizations: 1) pruning based on the fields that the predicate accesses (to monitor the accesses, Korat instruments the predicate and all methods that the predicate transitively invokes) and 2) generating only non-isomorphic candidates. These optimizations speed up the search without compromising its correctness.

Most practical predicates are deterministic, i.e., given identical inputs, any two executions of the predicate are identical and produce identical result. Korat also handles predicates that are non-deterministic either only in the execution (i.e., given identical inputs, two executions may be different

```
Finitization finSearchTree(int numNode,
    int minSize, int maxSize, int minInfo, int maxInfo) {
  Finitization f = new Finitization(SearchTree.class);
  ObjSet nodes = f.createObjects("Node", numNode);
  nodes.add(null);
  f.set("root", nodes);
  f.set("size", new IntSet(minSize, maxSize));
  f.set("Node.left", nodes);
  f.set("Node.right", nodes);
  f.set("Node.info", new IntegerSet(minInfo, maxInfo));
  return f;
}
Finitization finSearchTree(int scope) {
  return finSearchTree(scope, 0, scope, 1, scope);
}
```

Figure 4: Two finitizations for the `repOk` method.

but produce identical result) or even in the result (i.e., given identical inputs, two executions may produce different results). Let γ be an input for a predicate π. We write $\pi(\gamma)$ for the set of results that executions of π can produce for input γ; γ is *valid* iff $\pi(\gamma)=\{\texttt{true}\}$; γ is *invalid* iff $\pi(\gamma)=\{\texttt{false}\}$. Note that an input may be neither valid nor invalid.

Each candidate that Korat generates is an object graph with one root object (Section 6 shows how to introduce a class for root when generating several objects). Executing the same Java program from two isomorphic states cannot lead to observational difference in the execution. Thus, we define structure isomorphism based on object identity; two candidates are isomorphic iff the object graphs reachable from the root are isomorphic.

DEFINITION 1. *Let $O_1, \ldots, O_n$ be some sets of objects from n classes. Let $O = O_1 \cup \ldots \cup O_n$, and suppose that candidates consist only of objects from O (and primitive values), i.e., pointer fields of objects in O can either be `null` or point to other objects in O. Let P be the set consisting of `null` and all values of primitive types, such as `int`. Let $r \in O$ be a root object, and let $R_C(r)$ be the set of all objects reachable from r in C. Two candidates, C and C', are* isomorphic *iff there exists a permutation p on $O \cup P$ that is identity on P and that maps objects from O_i to objects from O_i for all $1 \leq i \leq n$, such that:*

$\forall o \in R_C(r).\ \forall f \in fields(o).\ \forall v \in O \cup P.$
$o.f = v$ *within* $C \Leftrightarrow p(o).f = p(v)$ *within* C'.

Isomorphism between candidates partitions the state space into *isomorphism partitions*. Since candidates and valid inputs are rooted and edge-labeled, it is easy to check isomorphism. However, Korat does not do that explicitly; instead, it avoids generating isomorphic valid inputs by not even considering isomorphic candidates.

In summary, Korat generates all non-isomorphic valid inputs within specified bounds; the search has these properties:

- **Soundness:** Korat generates no invalid input.
- **Completeness:** Korat generates at least one valid input from each isomorphism partition.
- **Optimality:** Korat generates at most one (valid) input from each isomorphism partition.

3.2 Finitization

To generate a finite state space for predicate's inputs, Korat uses a finitization, i.e., a set of bounds that limits the size of the inputs. The inputs can consist of objects from several classes, and the finitization specifies the number of objects for each of those classes. A set of objects from one class forms a *class domain*. The finitization also specifies a set of values for each field; this set forms a *field domain*, which is a union of several class domains.

Korat provides a `Finitization` class that allows finitizations to be written in Java. Korat automatically generates a finitization *skeleton* from the type declarations in the Java code. Testers can further specialize or generalize this skeleton. Figure 4 shows two finitizations for the example `repOk` method; invoking `finSearchTree(s)` creates a finitization for scope `s`. The `createObjects` method specifies that the input contains at most `numNode` objects from the class `Node`. The `set` method specifies a field domain for each field.

3.3 State space

Korat uses a finitization to construct a state space of predicate inputs. For example, consider `finSearchTree(3)` for inputs to `repOk`. Korat first allocates one `SearchTree` object that forms the `SearchTree` class domain and three `Node` objects that form the `Node` class domain. In order to systematically explore the state space, Korat orders the objects in these domains and during search uses indexes into these domains. This data can be represented with these classes:

```
class ClassDomain { // ordered class domain
  Object[] objects;
}
class ClassDomainIndex {
  ClassDomain domain;
  int index; // index into 'domain.objects' array
}
class ObjField { // field of an object from some domain
  Object object;
  Field field;
}
```

Korat next assigns a field domain to each field. In this example, there are $11 = 2 + 3 \cdot 3$ fields: the single `SearchTree` object has two fields (`root` and `size`) and the three `Node` objects have three fields each (`left`, `right`, and `info`). Each field domain is a sequence of class domain indexes, such that all values that belong to the same class domain occur consecutively. For example, the field domain for `root` has four elements, `null` and three `Node` objects, where `null` (as well as each primitive value) forms a class domain by itself. Therefore, the field domain for `root` is represented as `[null,<nd,0>,<nd,1>,<nd,2>]`, where `nd` is the class domain for `Node` objects.

Each *state* is a mapping from the object fields to the field domain indexes. The whole state space consists of all possible mappings, i.e., it is the Cartesian product of the field domains for all fields. In this example, the domains for `root`, `left`, and `right` have four elements, the domain for `size` has four elements, and the domain for `info` has three elements; the state space has $4 \cdot 4 \cdot (4 \cdot 4 \cdot 3)^3 = 1769472 > 2^{20}$ states. For `scope`$= n$, the state space has $(n+1)^{2(n+1)} \cdot n^n$ states.Each state encodes a *candidate* input that consists of the Java objects from the finitization; each field of these objects is set according to the field domain indexes in the state. The Korat search builds states for systematic exploring of the state space, and it builds candidates as inputs to the predicate. Because of the bijection between states and candidates, we use terms "state" and "candidate" interchangeably. We define two states to be isomorphic iff the corresponding candidates are isomorphic.

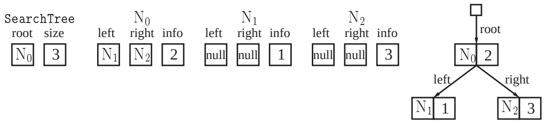

Figure 5: Candidate that is a valid SearchTree.

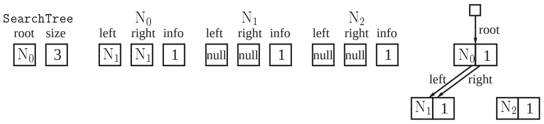

Figure 6: Candidate that is not a valid SearchTree.

Figure 5 shows an example candidate that is a valid binary search tree with three nodes. Not all candidates represent valid binary search trees. Figure 6 shows an example candidate that is not a tree; repOk returns false for this candidate. Assume that the field domains are ordered as follows: for left and right the same as for root (first null then Node objects), for size [0,1,2,3], and for info [Int(1),Int(2),Int(3)]. Then, the candidate in Figure 5 (Figure 6) corresponds to the state [1,3,2,3,1,0,0,0,0,0,2] ([1,3,2,2,0,0,0,0,0,0,0]).

3.4 Search

Figure 7 shows the pseudo-code of Korat's search algorithm. The search starts with the state set to all zeros. For each state, Korat first creates the corresponding candidate. Korat then executes the predicate to check the validity of the candidate. During the execution, Korat monitors the fields that the predicate accesses and maintains a stack of fields ordered by the first time the predicate accesses the corresponding field, i.e., whenever the predicate accesses a field not already on the stack, Korat pushes the field on the stack. As an illustration, consider that the stack is empty and Korat invokes repOk on the candidate shown in Figure 6. In this case, repOk accesses only the fields [root,N$_0$.left,N$_0$.right] (in that order) before detecting a cycle and returning false. Thus, the stack consists of only those three fields.

If the predicate returns true, Korat adds the current state to the set of valid inputs. It also makes sure that all reachable fields are on the stack, so that successive iterations generate all (non-isomorphic) states that have the same values for the accessed fields as the current state.

Korat then generates the next state backtracking on the accessed fields. Korat first increments the field domain index for the last field in the stack. If the index exceeds the domain size, Korat resets the index to zero and moves to the previous field in the stack, unless the stack is empty. Continuing with our example, the next candidate takes the next value for N$_0$.right, which is N$_2$ by the above order; the other fields do not change. This prunes from the search all $4^5 \cdot 3^3 = 27648$ states of the form [1,_,2,2,_,_,_,_,_,_,_] that have the (partial) valuation: root=N$_0$, N$_0$.left=N$_1$, N$_0$.right=N$_1$. Intuitively, the pruning based on accessed fields does not rule out any valid data structure because repOk did not read the other fields, and it could have returned false irrespective of the values of those fields.

```
Set<Map<ObjField,int>>
koratSearch(Predicate pred, Finitization fin) {
  Map<ObjField,ClassDomainIndex[]> space = fin.getSpace();
  Object root = fin.getRootObject();
  Set<Map<ObjField,int>> inputs = new Set();
  Stack<ObjField> stack = new Stack();
  Map<ObjField,int> state = new Map();
  foreach (ObjField f in fin.getObjFields())
    state[f] = 0;
  do {
    // create candidate input
    foreach (ObjField f in fin.getObjFields()) {
      ClassDomainIndex cdi = space[f][state[f]];
      f.set(cdi.domain.objects[cdi.index]);
    }
    // execute "pred(root)" and update "stack"
    boolean result = observeExecution(pred, root, stack);
    // if state is valid, add it to the valid inputs
    if (result) inputs.add(state);
    // if *not* optimizing, add other fields to the stack
    if (!PRUNING || result) {
      // add all reachable fields not already in stack
      foreach (ObjField f in reachableObjFields(root))
        if (!stack.contains(f)) stack.push(f);
    }
    // backtrack
    while (!stack.isEmpty()) {
      ObjField f = stack.top(); // field on the top of stack
      if (ISOMORPHISM_BREAKING) {
        int m = -1; // $m_f$ from the first Korat paper
        ClassDomain d = space[f][state[f]].domain;
        // a straightforward way to compute 'm_f'
        foreach (ObjField fp in stack.withoutTop())
          if (space[fp][state[fp]].domain == d)
            m = max(m, space[fp][state[fp]].index);
        // if an isomorphic candidate would be next...
        if (space[f][state[f]].index > m)
          // ...skip to the end of domain
          while (state[f] < space[f].length - 1 &&
                 space[f][state[f] + 1].domain == d)
            state[f]++;
      }
      if (state[f] < space[f].length - 1) {
        state[f]++; // increment this field
        break; // stop backtracking
      } else {
        state[f] = 0; // reset this field
        stack.pop(); // keep backtracking
      }
    }
  } while (!stack.isEmpty()); // end do
  return inputs;
}
```

Figure 7: Pseudo-code of Korat's search algorithm.

We next present the isomorphism-breaking optimization. Recall that a state is a mapping from object fields to field domain indexes that have a natural order. Additionally, each stack imposes a (partial) order on the fields. Together, these orders induce a (partial) lexicographic order on the states. Korat generates inputs in this lexicographical order. Moreover, Korat avoids generating states that are isomorphic to each other: for each isomorphism partition, Korat generates only the lexicographically smallest state in that partition. Conceptually, Korat avoids generating isomorphic states by incrementing some field domain indexes by more than one.

For the field f on top of the stack, Korat finds m, the maximum class domain index of all fields fp that are deeper on the stack and have the same domain as f (or -1 if there is no such fp). (The actual implementation uses caching to compute m.) For the example state from Figure 6 with the stack as discussed above, m=1 for f=N$_0$.right. When backtracking on f, Korat checks if the field domain index for f is greater than m. If it is, Korat increments the field domain index of f to the end of the current class domain for f.

4. CORRECTNESS

This section presents the correctness of the Korat search algorithm. We first state assumptions and prove properties of the predicates that the search operates on. The output of the search is a set of predicate inputs. This set depends on the results that the predicate returns for the candidate inputs. If the predicate is non-deterministic in result, Korat can generate different sets of inputs for the same predicate and finitization. We prove that all those sets satisfy soundness, completeness, and optimality properties.

Several issues complicate proving Korat's correctness:

- **Encoding:** For efficiency, Korat encodes Java objects using integers (that index into the finitization).
- **Field ordering:** Korat searches the state space based on a *dynamic* order in which the predicate executions access the fields.
- **Isomorphism optimization:** Korat generates only non-isomorphic inputs, but does so without explicitly checking isomorphism of pairs of inputs.
- **Non-deterministic predicates:** Korat generates all inputs for which the predicate always returns `true` and no input for which the predicate always returns `false`; Korat may or may not generate the other inputs.

We prove that the Korat search is correct for all predicates $\pi \in \Pi$ whose executions satisfy the following conditions:

1. Each execution terminates (returning `true` or `false`).
2. Each field that the execution accesses is: 1) reachable from the input or 2) a field of some object that the predicate locally allocates.
3. No execution invokes `System.identityHashCode` method.

The user of Korat needs to write the predicate so that Condition 1 holds. Condition 2 is easy to establish: the predicate should not access global data (through instance fields), but only the input. Condition 3 is easy to statically check.

We first show that the execution of a predicate does not depend on the fields that the predicate does not access.

LEMMA 1. *Consider two candidates γ and γ' that have identical values for all fields from some set ϕ. If an execution of π with input γ accesses only fields from ϕ before returning a result, then there exists an execution of π that acccesses the same fields and returns the same result.*

PROOF. Easy induction on the length of execution of the predicate for γ. As the witness execution for γ', choose the execution that makes the same steps as the execution for γ, i.e., the execution that for γ' makes the same non-deterministic choices as the execution for γ. At each corresponding step of these two executions, the states have identical values for all fields that are in ϕ, because no step accesses a field not in ϕ. □

The following is a simple corollary.

COROLLARY 1. *If two candidates have identical values for more fields than in ϕ, these two candidates have the same set of executions.*

We show that isomorphic inputs return the same result.

LEMMA 2. *For all isomorphic γ and γ'*, $\pi(\gamma) = \pi(\gamma')$.

PROOF. We need to show that for each execution of π with input γ, there exists an execution of π with input γ' such that the two executions generate the same result. Proof proceeds by induction on the length of execution for γ. As the witness execution for γ', choose the execution that makes the same steps as the execution for γ. By Condition 2, no execution of π accesses a field from the finitization that is not reachable from the root object (the input to the predicate). Thus, at each corresponding step of these two executions, the states have the same values for all fields 1) reachable from the root object or 2) belonging to the objects locally allocated. Further, by Condition 3, no step depends on the object identity, so the states are isormophic for these fields and in the final state, the executions return the same result. □

It follows that isomorphic inputs have the same (in)validity, which in fact allows Korat to consider as a candidate only one representative from each isomorphism partition.

We next consider properties of the search. We introduce some additional notation. We use σ to denote the value of the `stack` of fields that the predicate executions access. We write len(σ) for the length of the stack σ and $\sigma(i)$ for the field at offset i, where 0 is the offset at the bottom, and len(σ) $- 1$ is the offset at the top. We define a partial order between states based on lexicographic order of values in the stack. Let Σ and Σ' be two states. We say that Σ is σ-smaller than Σ', in notation $\Sigma <_\sigma \Sigma'$, iff $\exists i <$ len(σ). $\Sigma'(\sigma(i)) < \Sigma(\sigma(i)) \wedge \forall i' < i.\ \Sigma(\sigma(i')) = \Sigma'(\sigma(i'))$.

THEOREM 1 (SOUNDNESS). *Korat does not generate an invalid input for any predicate (even if not from Π).*

PROOF. By contradiction; suppose that Korat generates an invalid input γ for some predicate π. It means that all executions of π for input γ return `false`. However, the algorithm in Figure 7 generates γ (i.e., adds γ to the set `inputs`) only if an execution of π returns `true`. Contradiction! □

A search is complete if it generates at least one valid input from each isomorphism partition. To prove completeness of Korat, we consider Korat*, the Korat search with the parameter `PRUNING` set to `false`.

LEMMA 3. *If Korat* is complete for some predicate from Π, then Korat is also complete for that predicate.*

PROOF. Assume that Korat* is complete, i.e., it always generates at least one valid input from each isomorphism partition. An input is generated if it is executed as a candidate input, and the execution returns `true`. For a valid input, all executions return `true`, so a valid input is generated if it is executed as a candidate. Since Korat* generates at least one valid input from each isomorphism partition, it also considers as a candidate at least one valid input from each isomorphism partition. Due to pruning, Korat considers less candidates than Korat*. We will show that Korat still considers as a candidate at least one valid input from each isomorphism partition. Thus, Korat generates at least one valid input from each isomorphism partition, i.e., Korat is complete.

By contradiction, suppose that Korat does not consider as a candidate any valid input from some isomorphism partition. Since Korat* considers such an input, it must be that

Korat prunes this valid candidate input γ. This pruning occurs after the predicate executes some candidate input γ' and returns `false`. Let the `stack` after the execution of γ' be σ. Before returning `false`, this execution has accessed only (some of the) fields from σ. Further, it is easy to show that Korat* prunes only candidate inputs that have the same valuation for all fields in σ as γ', i.e., $\forall f \in \sigma.\gamma[f] = \gamma'[f]$. By Corollary 1, there exists an execution of γ that returns `false`. This contradicts the assumption that γ is valid, i.e., all executions for input γ return `true`. □

LEMMA 4. *Korat* is complete for all predicates from* Π.

PROOF. (Sketch) We need to show that for Korat* generates at least one valid input from each isomorphism partition. It is sufficient to show that Korat* considers as a candidate at least one valid input from each isomorphism partition.

The proof proceeds by induction on the number of considered candidates, i.e., the number of iterations of the main loop of Korat*. Each iteration consists of a predicate execution with potential adding of fields to the stack, and backtracking with isomorphism breaking. Let σ and Σ be the values of `stack` and `state`, respectively, after backtracking. Let Γ be the set of candidates considered up to that iteration. Recall the ordering between states $\Sigma <_\sigma \Sigma'$. Let s be $\{\Sigma' | \Sigma <_\sigma \Sigma'\}$ if σ is not empty and the set of all candidates if σ is empty. An easy induction can show that Γ contains at least one representative from each isomorphism partition that has a representative in s. Since the search terminates when the stack becomes empty, it follows that the search considers at least one candidate input from each isomorphism partition of candidates. □

THEOREM 2 (**COMPLETENESS**). *Korat is complete for all predicates from* Π.

PROOF. Follows from Lemma 3 and Lemma 4. □

We finally prove optimality.

THEOREM 3 (**OPTIMALITY**). *Korat is optimal for all predicates from* Π.

PROOF. Let Γ be the set of candidate inputs for which Korat executes the predicate. Since the result of Korat is a subset of Γ, it suffices to show that the search executes the predicate for at most one input from each isomorphism partition. By contradiction; suppose that there are two isomorphic distinct candidate inputs, γ and γ'. Let p be a permutation between these candidates. Consider the stacks, σ and σ', after the backtracking for candidates γ and γ', respectively. Let σ_0 be the common prefix for these two stacks. There are two cases:

- For some field in σ_0, the candidates have different values. Let i be the index such that

  $$\gamma[\sigma_0(i)] \neq \gamma'[\sigma_0(i)]. \tag{1}$$

  and

  $$\forall i' < i.\gamma[\sigma_0(i)] = \gamma'[\sigma_0(i)]. \tag{2}$$

  Since γ and γ' are isomorphic,

  $$\forall i \in \mathrm{len}(\sigma).p(\gamma[\sigma_0(i)]) = \gamma'[\sigma_0(i)]. \tag{3}$$

  From 2 and 3, we have that p is identity for all $i' < i$:

  $$\forall i' < i.p(\gamma[\sigma_0(i')]) = \gamma[\sigma_0(i')]. \tag{4}$$

  We next consider two cases based on the value for the i-th field already appearing before in the stack:

  $$\exists i' < i.\gamma[\sigma_0(i)] = \gamma[\sigma_0(i')]. \tag{5}$$

  - If (5) holds, let i' be the previous index. We have $p(\gamma[\sigma_0(i)]) = p(\gamma[\sigma_0(i')])$. From (4), we have that $p(\gamma[\sigma_0(i)]) = \gamma[\sigma_0(i')]$. Further, again from (5), we have $p(\gamma[\sigma_0(i)]) = \gamma[\sigma_0(i)]$ and then from (3) $\gamma'[\sigma_0(i)] = \gamma[\sigma_0(i)]$, which contradicts (1).
  - If (5) does not hold, then $\forall i' < i.\gamma[\sigma_0(i)] \neq \gamma[\sigma_0(i')]$. Let $\mathrm{maxIndex}(\gamma, \sigma_0, i)$ be the unique value m that the search computes in `m`. For field $\sigma_0(i)$, the search generates values up to m. Since $\gamma[\sigma_0(i)]$ is different than $\gamma[\sigma_0(i')]$ for all $i' < i$, it means that $\gamma[\sigma_0(i)] = m$, and this is the only such candidate. Thus, $\gamma'[\sigma_0(i)] = m$ also, and again $\gamma[\sigma_0(i)] = \gamma'[\sigma_0(i)]$; contradiction!

- For all fields in σ_0, the candidates have the same value. There are two cases depending whether σ_0 is the same as σ or σ'.
  - If σ_0 is the same as one of the stacks, assume w.l.o.g. that $\sigma = \sigma_0$. Consider the smallest index i such that $\gamma[\sigma'(i)] \neq \gamma'[\sigma'(i)]$; there must be such a field since $\gamma \neq \gamma'$. (More precisely, this value is not `0` in γ'; it is `0` in γ, because it is not in the stack for γ.) Since the search generates γ', it means that the for all indices i of σ', the values of the fields $\gamma'[\sigma'(i)] \leq \mathrm{maxIndex}(\gamma', \sigma', i)$. Further, for all $i' \leq i$, fields $\gamma'[i']$ are reachable in both γ and γ'. However, one of those fields, $\sigma'(i)$, has different values in γ and γ': $\gamma[\sigma'(i)] < \gamma'[\sigma'(i)] \leq \mathrm{maxIndex}(\gamma', \sigma', i)$. Thus, γ and γ' are not isomorphic; contradiction!
  - If σ_0 is not the same as any of the stacks, then the two stacks have a different field right after σ_0. Assume w.l.o.g. that the search first generates the stack σ. The only way to change the value of the field after σ_0, say f, is during backtracking if the search tries all possibilities for f and then backtracks on the previous field, the last in σ_0. This backtracking, however, changes the value of this last field in σ_0. Therefore, σ_0 cannot be the common prefix for σ and σ', if the candidates have the same values for all fields in σ_0. Contradiction!

This cover all the cases, and in each of them gives a contradiction if we suppose that the search considers two distinct, isomorphic candidate inputs. Hence, Korat is optimal. □

5. INPUT PROPERTY EXPLOITATION

We have found certain checks to be common in class invariants (`repOk` methods), e.g., that a linked data structure is acyclic along some fields or that an array has all elements different or ordered. Similar checks are also present in combinatorial optimization problems, and they are supported in libraries or languages such as Comet [38].

We have extended Korat with a library of *dedicated generators* that make it easier to write specifications and also enable faster generation of valid inputs. The library provides methods for the common checks. The specifications, or any other code, can use these methods; in regular execution, they behave like other Java methods. However, when Korat generates valid inputs, it uses the special knowledge about the methods to further optimize its search.

For example, `repOk` in `SearchTree` invokes the method `isAcyclic` that checks that the nodes reachable from the `root` field form a tree along the `left` and `right` fields. Appendix A shows a way to write `isAcyclic` in about 20 lines of code. Instead, we could just use the library method `korat.isTree(root, new String[]{"left", "right"})`. This method is parametrized over the root node and the names of the fields. Given a root node, `isTree` checks that the reachable nodes form a tree; essentially, it means that no node repeats in the traversal of the nodes reachable from the root. The search for the library method is implemented to take into account this fact.

When Korat generates an input that satisfies `isTree` along some fields, it does not try all (non-isomorphic) possibilities for those fields. Instead, each field is either `null` or points to a node that is not already in the tree. In our example `finSearchTree(s)`, this reduces the number of possibilities for one field from s+1 to 2. In the library, the implementation of `isTree` uses the basic dedicated generator `korat.isIn(field, set)` that, while searching, assigns to the `field` only the values from the `set`, and while checking, checks that the value of `field` is in the `set`.

The library includes the *basic* dedicated generators for checking the following properties: 1) a value is in a set; 2) two values are equal; 3) a value is less/greater than another value; and 4) a value is of a certain class (`instanceof`). The library also includes *combinators* that allow creating complex generators for checking: 1) negation, 2) conjunction, and 3) disjunction. Finally, the library includes several higher-level generators, implemented using basic generators and combinators, which check structural constraints such as acyclicity or that elements of an array are sorted.

It is easy to add new generators; in theory, we could even add for each data structure that we consider a special-purpose generator that generates all valid inputs without any backtracking. For example, such a generator for red-black trees was developed and used for testing in [3]. However, we do not do that; the library that we use in the experiments has only generators that are applicable for several data structures. In practice, we do not expect Korat users to extend the library, but instead to use Korat as a general-purpose search.

6. TESTING USING KORAT

This section describes how Korat automates test-input generation and correctness checking for Java programs. Korat constructs appropriate predicates from the specification and the code under test. We describe how to apply Korat to black-box (Section 6.1) and white-box (Section 6.2) testing.We also describe how to test sequences of method calls (Section 6.3). Sequences often arise in the context of algebraic specifications [23], for example insertion of an element followed immediately by its deletion from a structure leaves the structure contents unchanged. The sequences can also describe correctness properties at an abstract level, as in model-based specifications [47]; Korat can test such sequences using appropriate abstraction functions [35]. Not only Korat enables testing sequences, it also enables *generating* sequences of interest. This allows Korat to generate tests that use API calls to construct input objects in addition to tests that directly represent concrete object graphs.

```
class SearchTree_remove {  // inputs to "remove"
    SearchTree This;       // (implicit) "this" parameter
    Comparable info;       // "info" parameter

    // for black-box testing of "remove"
    boolean removePre() {  // precondition for "remove"
        return This.repOk();
    }

    // for white-box testing of "remove"
    boolean removeFail() { // failure for "remove"
        if (!removePre()) return false;
        try { // invoke "remove" with JML assertions
            This.remove(info);
        } catch (JMLAssertionException e) {
            return true;   // postcondition not satisfied
        }
        return false;
    }
}
```

Figure 8: Class for inputs to the `remove` method.

6.1 Black-box testing

In black-box testing, Korat tests a method without considering the method's code. Korat exhaustively generates inputs that satisfy the method precondition, executes the method on each of the inputs and checks the output using a *test oracle*. To generate test inputs for a method m, Korat first constructs a Java class corresponding to the m's inputs and a predicate corresponding to the m's precondition. Korat then generates valid inputs for that predicate; each of these inputs corresponds to a valid test input for m. For the `remove` method from Section 2, the corresponding class and the predicate `removePre` are shown in Figure 8. The predicate simply invokes `repOk` on the (implicit) `this` parameter of `remove`; the parameter `info` is unconstrained.

After generating all valid test inputs for a method, Korat invokes the method on each input and checks each output with a test oracle. A simple test oracle could check partial correctness of a method by invoking `repOk` in the post-state to check if the method preserves its class invariant. If the result is `false`, the method under test is incorrect, and the input provides a concrete counterexample.

Korat currently uses the JML tool-set [33] to automatically generate test oracles from method postconditions (and method assertions in general), as in the jmlunit framework [8]. The JML tool-set translates JML postconditions (and assertions) into runtime Java assertions. If an execution of a method violates such an assertion, an exception is raised. Test oracle catches these exceptions and reports correctness violations. These exceptions are different from the exceptions that the method specification allows, and Korat leverages JML to check both normal and exceptional behavior of methods. More details on the JML tool-set and translation can be found in [33].

6.2 White-box testing

In white-box testing, Korat tests a method considering the method's code. To test a method m, Korat first constructs a predicate corresponding to the negation of m's correctness. If a valid input is found for this predicate, m is incorrect, and the input provides a counterexample. For the `remove` method, the corresponding predicate `removeFail` is shown in Figure 8. This predicate first invokes `removePre`; if it is not satisfied, the input is not a valid test input for `remove` and cannot be a counterexample. If the input is valid, `remove` is executed, together with the JML-translated assertions. If this execution raises a JML exception, `remove` failed to satisfy its specification.

The difference between predicates for white-box and black-box testing is in the invocation of the method under test; in our example, `removeFail` invokes `remove`, but `removePre` does not. It means that for generating valid inputs to `removeFail`, Korat instruments `remove`, among other methods, and monitors the accesses that `remove` makes to the candidate. This by itself makes one execution of `remove` slower. However, it "opens" the body of `remove` for the optimizations that Korat performs to prune the search. In general, this can significantly reduce the time to test the method.

6.3 Sequences of method calls

We next consider testing a sequence of method calls. It is straightforward to translate the problem of testing a fixed sequence to the problem of testing one method. As an illustration, consider the following two example sequences that specify properties of `remove`: 1) the sequence `t.add(e).remove(e).equals(t)` that arises in an axiom in an algebraic specification and 2) the sequence

```
  JMLObjectSet sPre = t.abstract();
  t.remove(e);
  JMLObjectSet sPost = t.abstract();
  return sPost.equals(sPre.delete(e));
```

that arises in a model-based specification for `remove`, and states that each tree is abstracted into a set and `remove` from a tree commutes with removing the element from the set. (This form of specification can be expressed in JML using `model` fields.)

The sequences are translated respectively into the methods `axiom` and `implements`:

```
/*@ requires t.repOk();
  @ ensures \result == true; */
boolean axiom(SearchTree t, Comparable e) {
    return t.add(e).remove(e).equals(t);
}

/*@ requires t.repOk();
  @ ensures \result == true; */
boolean implements(SearchTree t, Comparable e) {
    JMLObjectSet sPre = t.abstract();
    t.remove(e);
    JMLObjectSet sPost = t.abstract();
    return sPost.equals(sPre.delete(e));
}
```

These methods can now be tested in the same way that `remove` is tested, either in black-box or white-box mode.

Korat not only enables testing sequences of method calls, it also enables *generating* sequences of interest. The user needs to build a representation of desired sequences and a predicate that defines their validity; generating a valid input to the predicate then provides a sequence with desired property. Clearly, Korat can generate all such sequences (up to a given length). Each such sequence corresponds to a common test input that consists of method calls; this contrasts with the typical use of Korat to generate inputs as concrete object graphs. However, the ability to generate sequences, allows Korat to discover a sequence that builds a given object graph. This enables Korat to output counterexamples (or other inputs of interest) as sequences of method calls.

7. MUTATION TESTING

This section presents design and implementation of Ferastrau, a tool for mutation testing of Java programs. *Mutation testing* is a criterion for assessing the quality of a set of test inputs [15, 41]. There are mutation testing tools for several languages, such as Mothra [31] for Fortran and Proteum [13] for C. We have implemented Ferastrau for Java; to the best of our knowledge, the first mutation-testing tool for Java. Mutation testing proceeds in two steps.

In the first step, a set of *mutant*s is generated from the original program by applying *mutation operators* to perform one or more syntactic modifications, e.g., replacing a variable with another variable (of a compatible type), say `n.left` with `n.right`. Mutation operators model typical software faults; the operators were characterized [2,30,31,42] for several languages, including Java. Section 7.1 presents mutant generation in Ferastrau.

In the second step, the original program and each mutant are executed on each input and the corresponding outputs are compared. If a mutant generates an output different from the original program, the test input is said to *kill* the mutant. For a given set of inputs, the rate of mutant killing is the ratio of the number of killed mutants to the total number of mutants. Section 7.2 presents how Ferastrau executes mutants and compares the outputs.

7.1 Mutant generation

We have implemented mutant generation by changing the Sun's `javac` compiler. Ferastrau performs a source-to-source translation: it parses each class of the original program into an abstract syntax tree, applies some mutation operators to the trees, and outputs the source of the mutants. Ferastrau applies the following mutation operators:

- Mutate a Java operator to another operator (of the same type), e.g., '+' to '-', '==' to '`!=`', '<' to '<=' etc.
- Mutate a variable to another variable (of a compatible type), e.g., a local variable `i` to `j` or an instance variable `n.left` to `n.right`.
- Mutate an invocation of a method to another method (of a compatible signature). (Ferastrau does not replace some special methods, such as `notify`; programmers typically do not make such mistakes.)

benchmark	“target” methods	some “helper” methods	# ncnb lines	# branches	# mutants
SearchTree	add, remove	contains	85	20	272
DisjSet	union, find	compressPath	29	8	243
HeapArray	insert, extractMax	heapifyUp, heapifyDown	51	9	274
BinomialHeap	insert, extractMin union, delete	contains, decrease merge, findMin	182	33	292
FibonacciHeap	insert, extractMin union, delete	contains, decrease cascadingCut, cut, consolidate	171	31	297
LinkedList	add, remove, reverse	contains, ListIterator.next	102	16	244
SortedList	insert, remove sort, merge	contains	176	29	231
TreeMap	put, remove	get, fixAfterInsertion containsKey, fixAfterDeletion rotateLeft, rotateRight	230	47	293
HashSet	add, remove	contains, HashMap.containsKey HashMap.put, HashMap.remove HashMap.rehash	113	20	244
AVTree	lookup	extract	199	26	205

Table 1: Benchmarks and target methods. Each benchmark is named after the main class; Korat generates data structures that also contain objects from other classes. Korat generates inputs and checks outputs for the target methods, thereby also testing helper methods. We tabulate the number of non-comment non-blank lines of source code in all those methods, the number of branches, and the number of mutants generated by Ferastrau.

The above operators modify only the code of methods, and not classes, i.e., do not add/remove a method or a field. These operators correspond to subtle mistakes that manifest only for non-trivial inputs, as the results in Section 8.3 show. It is easy to add new operators to Ferastrau to test different kind of mistakes.

Ferastrau generates mutant classes that have the same name as the corresponding original classes. For reasons explained below, Ferastrau provides two approaches: 1) generate the same classes with both the original program and the mutants or 2) generate different classes. Suppose that the original programs contains `temp.right` that is to be mutated to `left.right`. The first approach uses *metamutants* [51]: the mutations are guarded by boolean variables that are appropriately set during mutant execution; it generates one class with `(MUT ? left : temp).right`. The second approach simply generates `left/*temp*/.right` in another class.

7.2 Mutant execution

After generating the mutants, Ferastrau uses a set of test inputs to perform mutation testing. Our experiments use inputs generated by Korat. Ferastrau executes the original program and the mutants for each input and compares their respective outputs. Ferastrau assumes that the original program terminates for all test inputs; mutation testing tools for other languages [13, 31] make the same assumption. Since Ferastrau operates on Java and has to handle potentially large number of inputs, additional questions arise:

- How to compare outputs and name mutated classes?
- Whether to execute the original program and the mutants in a single run or in separate runs?
- How to handle non-termination and exceptional termination of the original program and the mutants?

We next describe how Ferastrau addresses these questions and then list the criteria that Ferastrau uses to kill a mutant.

Recall that the “output” of a method refers to both the return value and the objects in the post-state. Comparison is easy when these are primitive values, but the objects can represent complex structures. Ferastrau by default uses `equals` methods to compare outputs, following Java convention of using `equals` for equality comparisons of objects. This allows comparisons based on *abstract* values; for example, two binary search trees that implement sets may be structurally different at the *concrete* level of the implementation, but if they represent the same set, they are equal according to the `equals` method. The use of `equals` requires that Ferastrau generates mutant classes that have the same name as the corresponding original classes.

Ferastrau executes the original program and the mutants in a single run; otherwise, it would need to serialize all the outputs, which could produce very large files for inputs exhaustively generated by Korat. Recall that Ferastrau can generate metamutants or generate the original program and the mutants in different classes. If Ferastrau uses metamutants, the guarding boolean variables slow down the execution. This approach is better for small code with large data. If Ferastrau generates different classes, it needs to execute several classes with the same name in a single Java Virtual Machine (JVM). Ferastrau then uses a different `ClassLoader` [50] for the original program and each mutant, and uses serialization through a buffer in memory to compare objects. This approach is better for large code with small data.

Ferastrau assumes that the original program terminates for all test inputs, either normally or exceptionally. These exceptions are allowed by the specification, so they are not errors. Ferastrau handles non-termination of mutants by running them in a separate thread with a time limit for execution. The limit is set to $T_m = 10T_o + 1sec$, where T_o is the time the original program runs for that input. We have found these constants sufficient to account for fluctuations in the execution time of Java programs, e.g., due to garbage collection. (The minimum of $1sec$ is necessary, at least for Sun’s Java 2 SDK1.3.x JVMs, when the mutant raises an exception, since JVM takes some extra time in those situations.) The mutants can terminate either normally or exceptionally. Ferastrau catches all exceptions (in Java, all `Throw-`

		generation					checking			
		gen.	ded.	spec. coverage		#	time	code coverage		mutants
benchmark	scope	[sec]	[sec]	st. [%]	br. [%]	inputs	[sec]	st. [%]	br. [%]	killed [%]
SearchTree	6	1.39	0.52	94.74	96.67	8772	0.46	100.00	100.00	99.26
	7	9.03	2.19	94.74	96.67	41300	1.25	100.00	100.00	99.26
DisjSet	4	0.29	0.31	100.00	100.00	18280	0.43	100.00	100.00	95.06
	5	10.91	9.87	100.00	100.00	1246380	19.93	100.00	100.00	95.06
HeapArray	6	0.90	0.71	90.00	92.86	118251	1.88	100.00	100.00	96.35
	7	7.09	6.21	90.00	92.86	1175620	17.58	100.00	100.00	96.71
BinomialHeap	6	3.30	2.35	97.67	98.00	159642	4.61	100.00	100.00	95.89
	7	35.60	28.06	97.67	98.00	2577984	75.96	100.00	100.00	96.91
FibonacciHeap	4	1.22	0.90	97.78	98.28	34650	1.08	95.70	98.39	81.48
	5	14.14	12.94	97.78	98.28	941058	23.37	100.00	100.00	88.88
LinkedList	6	0.33	0.31	100.00	100.00	11741	0.48	90.57	84.38	99.59
	7	0.74	0.71	100.00	100.00	58175	1.54	90.57	84.38	99.59
SortedList	6	1.94	1.77	100.00	100.00	73263	2.57	92.50	89.66	97.40
	7	22.68	21.13	100.00	100.00	1047608	37.91	92.50	89.66	97.40
TreeMap	6	0.94	0.61	100.00	100.00	3924	0.38	100.00	91.49	89.76
	7	3.28	1.75	100.00	100.00	12754	0.73	100.00	91.49	89.76
HashSet	6	0.91	0.71	89.47	92.31	12932	0.62	100.00	100.00	91.80
	7	3.38	2.88	89.47	92.31	54844	1.55	100.00	100.00	92.21
AVTree	4	3.16	1.86	96.67	96.88	27734	8.36	94.12	92.31	91.21
	5	87.13	43.41	96.67	96.88	417878	134.51	94.12	92.31	93.65

Table 2: Korat's performance for test generation (with regular and dedicated generators), specification coverage (statement and branch), correctness checking, code coverage (statement and branch), and rate of mutant killing. All times are elapsed real times in seconds from the start of Korat to its completion.

`able` objects) that the executions raise. This allows Ferastrau to compare the outputs, even exceptional, as well as to catch all errors in the mutants, including the situations when the mutant runs out of stack or heap memory and JVM raises `StackOverflowError` or `OutOfMemoryError`.

Ferastrau uses the following criteria to kill a mutant:

- The mutant's output does not satisfy some class invariant (`repOk`), which is a precondition for `equals`.
- The mutant's output differs from the output of the original program; any of the outputs can be normal or exceptional.
- The mutant's execution exceeds the time limit.
- The mutant's execution runs out of memory.

8. EXPERIMENTAL RESULTS

This section presents the experiments that evaluate exhaustive testing. We first discuss the quality of generated test suites showing how the rate of mutant killing and coverage vary with the scope. We then discuss Korat's performance for test input generation and checking method correctness. We finally compare exhaustive testing with randomly selected test inputs.

8.1 Benchmarks

Table 1 lists the benchmarks and methods that we use to measure Korat's performance. We use Korat to generate inputs and check the correctness of outputs for the *target* methods. These methods implement the standard operations on their corresponding data structures [12]. Executing these methods also tests some *helper* methods because they are invoked either when executing the target methods or when checking their correctness (e.g., from postconditions).

`SearchTree` is presented in Section 2. `DisjSet` is an array-based implementation of the fast union-find data structure [12]; this implementation uses both path compression and rank estimation heuristics to improve efficiency. `HeapArray` is an array-based implementation of the heap (priority queues) data structure. `BinomialHeap` and `FibonacciHeap` are dynamic data structures that also implement heaps, but differ in complexity for certain operations [12].

`LinkedList` is the implementation of linked lists in the Java Collections Framework, a part of the standard Java libraries [50]. This implementation uses doubly-linked, circular lists. This benchmark is also representative for linked data structures such as stacks and queues. The elements in `LinkedList` are arbitrary objects; `SortedList` is structurally identical to `LinkedList`, but the elements are sorted. This benchmark is similar to the examples used in some shape analyses [34, 39]. `TreeMap` implements the `Map` interface using red-black trees [12]. `HashSet` implements the `Set` interface, backed by a hash table [12].

`AVTree` implements the *intentional name* trees that describe properties of services in the Intentional Naming System (INS) [1], an architecture for service location in dynamic networks. The original implementation of INS had bugs that we revealed with exhaustive testing [36] and corrected. We use the corrected version as the original program in these experiments, but (some of) the mutants have errors.

8.2 Mutation

For mutation testing, we use Ferastrau. We instruct Ferastrau to generate between 200 and 300 mutants for each benchmark, mutating the target methods and the helper methods they invoke, but not the helper methods that only specifications invoke. Table 1 shows the number of mutants for each benchmark. Table 2 shows the rate of mutant killing for several scopes. (The numbers for smaller scopes are in Appendix B.) For all benchmarks but `FibonacciHeap` and `TreeMap`, inputs in these scopes kill over 90% of the mutants.

We inspected a selection of the mutants that survived for these two benchmarks to detect if they are syntactically dif-

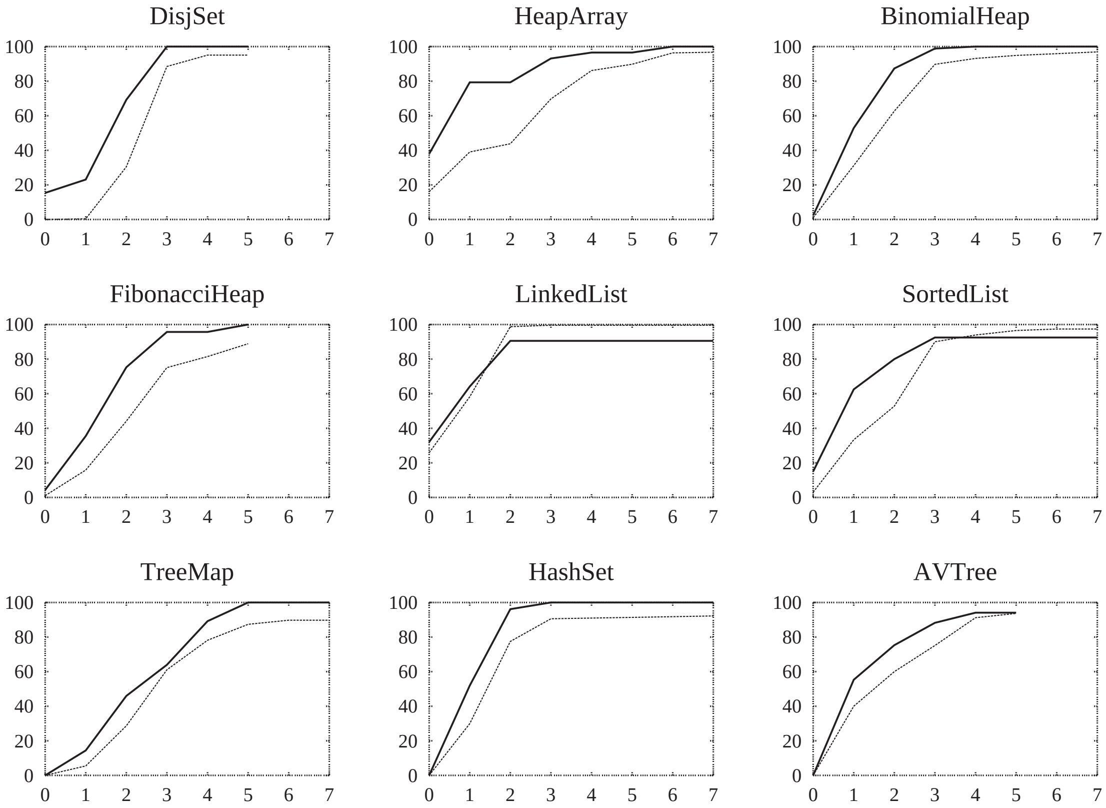

Figure 9: Variation of statement code coverage (thick line) and rate of mutant killing (thin line) with scope. For *all* benchmarks, Korat generates inputs that achieve the maximum coverage that is possible without directly generating inputs for helper methods.

ferent but *semantically* equivalent to the original program, and thus no input could kill them. Since we implemented `FibonacciHeap`, we were able to determine that all inspected mutants are indeed semantically equivalent. Due to the complexity of `TreeMap` and our lack of familiarity with it, we were not able to definitely establish the equivalence for all inspected mutants. However, we also tested surviving mutants for inputs of scope eight and all still survived increasing our confidence that they are indeed semantically equivalent.

8.3 Coverage

Table 2 also shows specification/code coverage. Since Korat uses executable specifications, we measure *specification coverage* [7] as code coverage for the predicate that corresponds to the method's precondition (e.g., `removePre`). We measure this coverage while Korat generates valid inputs for the predicate, i.e., valid test cases for the method. For most benchmarks, the tabulated scopes achieve complete coverage, both for statements and branches. It is not always 100%, because finitizations do not even put for fields some values that do not satisfy the predicate (e.g., `finSearchTree` does not put `null` for `info`). Specification coverage becomes complete for smaller scopes than code coverage (Appendix B) which means that specification coverage is a weaker criterion than code coverage.

Figure 9 shows graphs that relate scope with the statement coverage of code and the rate of mutant killing. The code coverage is measured for all target and helper methods, since they are all executed. For most benchmarks, Korat generates inputs that achieve 100% coverage, both for statements and branches. For other benchmarks, the coverage is not 100%, because no input for target methods could trigger some exceptional behavior of helper methods.

For example, the (target) `reverse` method for lists creates a `ListIterator` and invokes some (helper) methods on it. In general, these helper methods could raise exceptions, such as `ConcurrentModificationException` or `NoSuchElementException`, but the target methods never invoke the helper methods in such a way. In terms of JML specifications, the target methods invoke the helper methods in pre-states that satisfy the precondition for `normal_behavior`, and not for `exceptional_behavior`.

For all benchmarks, the minimum scope needed to achieve complete coverage is not sufficient to kill all mutants; increasing the scope increases the rate of mutant killing. It is well-known [5] that in general complete statement and branch coverage (or for that matter, any coverage criteria) does not guarantee absence of faults. Our experiments validate this for data structures. As an illustration, consider the following code snippet from `SearchTree.remove`:

```
Node temp = left;
while (temp.right.right != null) {
    temp = temp.right;
}
```

Suppose that the mutation changes only the loop body:

```
Node temp = left;
while (temp.right.right != null) {
    temp = left/*temp*/.right;
}
```

If the loop is executed zero or one times, the original program and the mutant have the same behavior. For trees with up to four nodes, the loop cannot execute more than once, but these trees achieve complete coverage for `remove`. However, for a tree with five nodes, the loop executes twice in the original program, while the mutant loops infinitely. (Recall that Ferastrau detects and kills such mutants.) Because of this, we do not measure the effectiveness of Korat by considering only the scope that achieves complete coverage.

8.4 Feasibility

Table 2 also shows Korat's performance for test generation and correctness checking. For each benchmark, the tabulated scope is mutation-adequate. We tabulate the time Korat takes to generate all valid test inputs (without and with dedicated generators) and to check the correctness of methods. All times are elapsed real times in seconds from the start of Korat to its completion, without the JVM initialization that takes around 0.5 seconds. We performed the experiments on a Linux machine with a 1.8GHz Pentium 4 processor using Sun's Java 2 SDK1.3.1 JVM.

Number of inputs that is generated is the sum of numbers of inputs for *all* target methods. Also, the generation and checking times are sums of times for all methods. We use Korat to separately generate inputs for each method. However, when two methods have the same precondition (e.g., `remove` and `add` for `SearchTree`), we could reuse the inputs and thus reduce the generation time. For scopes in Table 2, the size of the search space is between 2^{25} and 2^{250}. In all cases, Korat completes generation in less than two minutes, often in just a few seconds.

The use of dedicated generators reduces the generation times from a few percent to 75% (for `SearchTree`). Since dedicated generators have a higher overhead, their use sometimes increases the generation time, especially for very small scopes. As mentioned in Section 5, Korat library is not aimed at providing the most efficient generation for our benchmarks, but at providing generators that are applicable for many data structures. Furthermore, dedicated generators make it easier to write specifications for all our benchmarks.

The postconditions for all methods specify typical partial correctness properties; they require resulting data structures to be valid and, depending on the method, to contain or not to contain input elements. The checking times depend on the complexity of methods and their postconditions. Overall, the checking times are within the same magnitude as the corresponding generation times across all our benchmarks.

These results show that Korat can efficiently generate all inputs even for very large search spaces, primarily because the search pruning allows Korat to explore only a tiny fraction of these spaces. The key to effective pruning is backtracking based on fields accessed during `repOk`'s executions. Without backtracking, and even with isomorphism optimization, Korat would consider infeasibly many candidates. Isomorphism optimization further reduces the number of considered candidates, but it mainly reduces the number of valid inputs and thus checking time.

		random	exhaustive	
benchmark	scope	mutants killed [%]	scope-1	scope
`SearchTree`	7	99.26	=	=
`DisjSet`	5	95.06	=	=
`HeapArray`	7	95.99	<	<
`BinomialHeap`	7	95.10	<	<
`FibonacciHeap`	5	86.87	>	<
`LinkedList`	7	99.59	=	=
`SortedList`	7	96.40	<	<
`TreeMap`	7	89.08	<	<
`HashSet`	7	91.39	<	<
`AVTree`	5	93.17	>	<

Table 3: Comparison of exhaustive testing with randomly chosen test inputs. '=' means that both sets are equally good, '<' random is worse,'>' random is better.

It is important to note that Korat generates exactly the number of non-isomorphic data structures as given in the Sloane's On-Line Encyclopedia of Integer Sequences [46]. This increases our confidence that the implementation of Korat is correct; we proved that its algorithm is correct.

8.5 Randomness

Exhaustive testing generates all tests (within a certain size) that satisfy the input invariant. A natural question is what test selection strategy can be applied to reduce the size of an exhaustive test suite without sacrificing its quality. The simplest selection strategy is random sampling. We next evaluate its quality.

Consider one benchmark, and let $T(s)$ be the set of all (non-isomorphic) test inputs within scope s. From $T(s)$, we randomly chose a subset $R(s)$ whose cardinality is the same as the cardinality of $T(s-1)$. We then compare the quality of $R(s)$ against $T(s-1)$ and $T(s)$. For comparison, we use the rate of mutant killing, as this criterion most directly measures the quality of test suite in detecting faults. It is important to note that randomly chosen inputs are also generated with Korat; for complex data structures, it is not feasible to simply generate random inputs that satisfy the invariants.

Table 3 shows the comparison for several random samples. The randomly selected test suites give a lower rate of mutant killing in half of the benchmarks; only for `FibonacciHeap` and `AVTree`, the rate is higher for randomly selected inputs than for all inputs from the smaller scope. This means that exhaustive testing for all inputs within some scope can be more effective than random testing with bigger inputs. There may be, however, another test selection strategy, besides random, that can reduce the size of an exhaustive test suite without reducing its quality.

9. RELATED WORK

We first discuss other projects on specification-based testing. We then compare Korat with static analysis and model checking; Korat is related to them although it performs testing, i.e., dynamic analysis, because it does so exhaustively.

9.1 Specification-based testing

There is a large body of research on specification-based testing. An early paper by Goodenough and Gerhart [20] emphasizes its importance. Many projects propose techniques and tools that automate test case generation from specifications, such as Z specifications [16, 25, 49], UML statecharts [40], or ADL specifications [7, 44]. These specifications typically do not consider linked data structures with complex invariants.

Korat is reimplemented in the AsmL Test Generation tool (AsmLT) [18] that generates test cases from AsmL specifications [22]. Korat adds structure generation to generation based on finite-state machines [21]. AsmLT was successfully used for detecting fault in a production-quality XPath compiler [48]. Scalability of exhaustive testing tools does not depend as much on the complexity/size of the tested code as it depends on the complexity of inputs that the code operates on. This paper evaluates the effectiveness of exhaustive testing for data structures.

The TestEra framework [36] generates Java data structures from declarative specifications given in Alloy [27]. TestEra uses the SAT-based analysis of the Alloy tool-set [26] for test generation and correctness checking. Even though Alloy provides some non-isomorphic generation, for efficient enumeration TestEra requires users to manually write symmetry breaking predicates [29]. Also, TestEra requires the use of a specification language much different from Java. The experimental results [6] show that Korat provides faster test generation than TestEra (even when TestEra users manually add symmetry breaking).

Cheon and Leavens [8] describe jmlunit, an automatic translation of JML specifications into test oracles for JUnit [4], a popular framework for unit testing of Java modules. JUnit automates test execution and error reporting, but requires programmers to provide test inputs and test oracles. In jmlunit, the Cartesian product is directly used to generate test inputs, which cannot handle very large input spaces. Additionally, jmlunit does not generate complex inputs, but requires users to create and provide them. Korat further automates and optimizes generation of test inputs, thus automating the entire testing process.

There are many tools that produce test inputs from a description of tests. QuickCheck [9] is a tool for testing Haskell programs. It requires the tester to write Haskell functions that can produce valid test inputs; executions of such functions with different random seeds produce different test inputs. Korat differs in that it requires only an invariant that characterizes valid test inputs and then uses a general-purpose search to generate *all* valid inputs. DGL [37] and lava [45] generate test inputs from context grammars. They were used mostly for random testing, although they can also exhaustively generate test inputs. However, they do not consider inputs with complex invariants.

AETG [10] is a popular system for generating test inputs that cover all pair-wise (or n-wise) combinations of test parameters (that correspond to object fields in Korat). Using pair-wise testing is applicable when parameters are relatively independent. However, in Korat fields are dependent, and Korat can be viewed as an efficient approach to generate all inputs when n is the same as the number of parameters. Additionally, Korat takes into account isomorphism and generates only one input from each isomorphism partition.

9.2 Static analysis

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [17] uses a theorem prover to verify partial correctness of classes annotated with JML specifications. ESC can verify absence of errors such as null pointer dereferences, array bounds violations, and division by zero. However, tools like ESC do not verify properties of complex linked data structures.

Some recent projects attempt to address this issue. The Three-Valued-Logic Analyzer (TVLA) [34, 43] is the first system to verify that the list structure is preserved in programs that perform list reversals via destructive updates. TVLA has been used to analyze programs that manipulate doubly linked lists and circular lists, as well as some sorting programs. The pointer assertion logic engine (PALE) [39] can verify a large class of data structures that can be represented by a spanning tree backbone. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. TVLA and PALE are primarily intraprocedural, whereas Role Analysis [32] supports compositional interprocedural analysis and verifies similar properties.

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, but for inputs bounded by a given size.

Jalloy [28, 52] analyzes methods that manipulate linked data structures by first building an Alloy model of Java code and then checking it exhaustively with the Alloy tool-set [26]. This approach provides static analysis, but unsound with respect to both the size of input and the length of computation. Korat checks the entire computation and handles larger inputs and more complex data structures than in [28, 52]. Further, Korat does not require Alloy, but JML specifications.

9.3 Software model checking

There has been a lot of recent interest in applying model checking to software. JavaPathFinder [53] and VeriSoft [19] operate directly on a Java, respectively C, program and systematically explore its state to check correctness. Other projects, such as Bandera [11] and JCAT [14], translate Java programs into the input language of existing model checkers like SPIN and SMV . They handle a significant portion of Java, including dynamic allocation, object references, exceptions, inheritance, and threads. They also provide automated support for reducing program's state space through program slicing and data abstraction.

However, most of the work on applying model checking to software has focused on checking event sequences and not linked data structures. Where data structures have been considered, the purpose has been to reduce the state space to be explored and not to check the data structures themselves. Korat, on the other hand, checks correctness of methods that manipulate linked data structures.

10. CONCLUSIONS

This paper evaluated effectiveness of exhaustive testing for several implementations of data structures. We measured how statement coverage, branch coverage, and rate of mutant killing vary with scope. We used Korat and its extensions to perform exhaustive testing. The experimental results show that: 1) exhaustive testing within small scopes can achieve complete coverage and kill almost all of the mutants for data structure benchmarks; 2) mutation adequacy is a stronger criterion than code coverage; 3) Korat can be used effectively to generate inputs and check correctness for these scopes; and 4) exhaustive testing within some scope can be more effective than random testing with bigger inputs.

We particularly point out the feasibility of exhaustive testing for data structures. The community has been aware of the potential utility of generating structures that satisfy sophisticated invariants [3], but the problem has been considered to be too difficult to attempt to solve:

> "Trying to generate a random heap state, with a rats' [sic] nest of references, and then select those that represent queues, would be both difficult and hopeless in practice." [9, page 68]

Our results show that it is, in fact, perfectly feasible in practice to efficiently generate structures that satisfy not just the queue property, but arbitrary data structure invariants as specified by arbitrary pieces of code written in a standard programming language. These results, together with previous studies that used exhaustive testing to detect faults in real applications [36, 48], suggest that exhaustive checking within a selected scope [6,18,52] is a practical way to obtain a high-quality test suite for data structure implementations.

Acknowledgements

We thank Michael Ernst and an anonymous reviewer of our TestEra paper [36] for suggesting to measure how coverage varies with scope. We thank Alan Donovan, Michael Ernst, Daniel Jackson, Jelani Nelson, and several other researchers for asking about correctness of Korat and thus providing motivation to write the proof. We thank Chandrasekhar Boyapati, Daniel Jackson, Alexandru Sălcianu, and Viktor Kuncak for comments on an earlier draft and for helpful discussions on the correctness of Korat. This work was funded in part by NSF grant CCR00-86154.

11. REFERENCES

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, Dec. 1999.

[2] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Purdue University, West Lafayette, IN, 1989.

[3] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *Software Testing, Verification & Reliability*, 10(3):149–170, 2000.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

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

[6] 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.

[7] 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.

[8] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and junit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[9] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 65–77, 2002.

[10] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[11] 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.

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

[13] M. E. Delamaro and J. C. Maldonado. Proteum—A tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996.

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

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 4(11):34–41, Apr. 1978.

[16] 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.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[18] Foundations of Software Engineering, Microsoft

Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[19] 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, Jan. 1997.

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

[21] 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.

[22] Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[23] J. Guttag and J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[24] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[25] H.-M. Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, 1995.

[26] 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.

[27] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, Sept. 2001.

[28] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, Aug. 2000.

[29] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

[30] S.-W. Kim, J. Clark, and J. McDermid. Class mutation: Mutation testing for object oriented programs. In *FMES 2000*, Oct. 2000.

[31] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software–Practice and Experience*, 21(7):685–718, 1991.

[32] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Portland, OR, Jan. 2002.

[33] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).

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

[35] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[36] 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.

[37] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[38] L. Michel and P. V. Hentenryck. A constraint-based architecture for local search. In *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 83–100, 2002.

[39] 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.

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

[41] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, Oct. 2000.

[42] J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, George Mason University, Fairfax, VA, Oct. 1996.

[43] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[44] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, CA, Apr. 1994.

[45] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proc. 2nd conference on Domain-specific languages*, pages 1–13, 1999.

[46] N. J. A. Sloane, S. Plouffe, J. M. Borwein, and R. M. Corless. The encyclopedia of integer sequences. *SIAM Review*, 38(2), 1996. `http://www.research.att.com/~njas/sequences/Seis.html`.

[47] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[48] K. Stobie. Advanced modeling, model based test generation, and Abstract state machine Language

AsmL. http://www.sasqag.org/pastmeetings/asml.ppt, 2003.

[49] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

[50] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. http://java.sun.com/j2se/1.3/docs/api/.

[51] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation testing using mutant schemata. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, June 1993.

[52] M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[53] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

APPENDIX

A. FULL CODE FOR THE EXAMPLE

```
import java.util.*;
class SearchTree {
    Node root; // root node
    int size;  // number of nodes in the tree
    static class Node {
        Node left;  // left child
        Node right; // right child
        Comparable info; // data
    }

    /*@ normal_behavior // non-exceptional specification
      @     // precondition
      @     requires repOk();
      @     // postcondition
      @     ensures repOk() && !contains(info) &&
      @             \result == \old(contains(info));
      @*/
    boolean remove(Comparable info) {
        Node parent = null;
        Node current = root;
        while (current != null) {
            int cmp = info.compareTo(current.info);
            if (cmp < 0) {
                parent = current;
                current = current.left;
            } else if (cmp > 0) {
                parent = current;
                current = current.right;
            } else {
                break;
            }
        }
        if (current == null) return false;
        Node change = removeNode(current);
        if (parent == null) {
            root = change;
        } else if (parent.left == current) {
            parent.left = change;
        } else {
            parent.right = change;
        }
        return true;
    }

    Node removeNode(Node current) {
        size--;
        Node left = current.left, right = current.right;
        if (left == null) return right;
        if (right == null) return left;
        if (left.right == null) {
            current.info = left.info;
            current.left = left.left;
            return current;
        }
        Node temp = left;
        while (temp.right.right != null) {
            temp = temp.right;
        }
        current.info = temp.right.info;
        temp.right = temp.right.left;
        return current;
    }

    boolean repOk() {
        // checks that empty tree has size zero
        if (root == null) return size == 0;
        // checks that the input is a tree
        if (!isAcyclic()) return false;
        // checks that size is consistent
        if (numNodes(root) != size) return false;
        // checks that data is ordered
        if (!isOrdered(root)) return false;
        return true;
    }

    private boolean isAcyclic() {
        Set visited = new HashSet();
        visited.add(root);
        LinkedList workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
            Node current = (Node)workList.removeFirst();
            if (current.left != null) {
                // checks that the tree has no cycle
                if (!visited.add(current.left))
                    return false;
                workList.add(current.left);
            }
            if (current.right != null) {
                // checks that the tree has no cycle
                if (!visited.add(current.right))
                    return false;
                workList.add(current.right);
            }
        }
        return true;
    }

    private int numNodes(Node n) {
        if (n == null) return 0;
        return 1 + numNodes(n.left) + numNodes(n.right);
    }

    private boolean isOrdered(Node n) {
        return isOrdered(n, null, null);
    }

    private boolean isOrdered(Node n, Comparable min, Comparable max) {
        if (n.info == null) return false;
        if ((min != null && n.info.compareTo(min) <= 0) ||
            (max != null && n.info.compareTo(max) >= 0))
            return false;
        if (n.left != null)
            if (!isOrdered(n.left, min, n.info))
                return false;
        if (n.right != null)
            if (!isOrdered(n.right, n.info, max))
                return false;
        return true;
    }
}
```

B. EXPERIMENTAL RESULTS

		generation					checking			
		gen.	ded.	spec. coverage		#	time	code coverage		mutants
benchmark	scope	[sec]	[sec]	st. [%]	br. [%]	inputs	[sec]	st. [%]	br. [%]	killed [%]
`SearchTree`	1	0.06	0.01	57.89	60.00	4	0.06	38.46	40.00	26.10
	2	0.05	0.01	94.74	96.67	20	0.06	79.49	87.50	69.85
	3	0.07	0.10	94.74	96.67	90	0.07	87.18	92.50	79.77
	4	0.17	0.10	94.74	96.67	408	0.14	97.44	97.50	92.64
	5	0.38	0.25	94.74	96.67	1880	0.24	100.00	100.00	98.52
	6	1.39	0.52	94.74	96.67	8772	0.46	100.00	100.00	99.26
	7	9.03	2.19	94.74	96.67	41300	1.25	100.00	100.00	99.26
`DisjSet`	1	0.01	0.01	61.54	55.00	4	0.04	23.08	25.00	0.41
	2	0.01	0.01	100.00	95.00	30	0.09	69.23	68.75	30.45
	3	0.04	0.04	100.00	100.00	456	0.09	100.00	100.00	88.47
	4	0.29	0.31	100.00	100.00	18280	0.43	100.00	100.00	95.06
	5	10.91	9.87	100.00	100.00	1246380	19.93	100.00	100.00	95.06
`HeapArray`	1	0.01	0.01	80.00	85.71	16	0.04	79.31	66.67	39.05
	2	0.01	0.01	90.00	92.86	75	0.05	79.31	66.67	43.79
	3	0.02	0.02	90.00	92.86	396	0.09	93.10	83.33	69.70
	4	0.08	0.09	90.00	92.86	2240	0.17	96.55	88.89	86.13
	5	0.22	0.21	90.00	92.86	15352	0.38	96.55	94.44	89.78
	6	0.90	0.71	90.00	92.86	118251	1.88	100.00	100.00	96.35
	7	7.09	6.21	90.00	92.86	1175620	17.58	100.00	100.00	96.71
`BinomialHeap`	1	0.02	0.01	62.79	62.00	12	0.07	52.87	57.58	31.16
	2	0.03	0.02	93.02	94.00	54	0.08	87.36	84.85	62.67
	3	0.12	0.09	93.02	94.00	336	0.14	98.85	96.97	89.72
	4	0.40	0.30	97.67	98.00	1800	0.24	100.00	98.48	93.15
	5	0.81	0.65	97.67	98.00	16848	0.69	100.00	100.00	94.86
	6	3.30	2.35	97.67	98.00	159642	4.61	100.00	100.00	95.89
	7	35.60	28.06	97.67	98.00	2577984	75.96	100.00	100.00	96.91
`FibonacciHeap`	1	0.01	0.07	55.55	51.72	12	0.07	35.48	43.55	15.82
	2	0.03	0.03	91.11	93.10	108	0.09	75.27	80.64	44.10
	3	0.28	0.24	97.78	98.28	1632	0.24	95.70	98.39	75.08
	4	1.22	0.90	97.78	98.28	34650	1.08	95.70	98.39	81.48
	5	14.14	12.94	97.78	98.28	941058	23.37	100.00	100.00	88.88
`LinkedList`	1	0.01	0.01	100.00	100.00	15	0.08	64.15	68.75	58.19
	2	0.01	0.01	100.00	100.00	50	0.09	90.57	84.38	98.77
	3	0.03	0.03	100.00	100.00	169	0.12	90.57	84.38	99.59
	4	0.07	0.07	100.00	100.00	627	0.16	90.57	84.38	99.59
	5	0.18	0.18	100.00	100.00	2584	0.26	90.57	84.38	99.59
	6	0.33	0.31	100.00	100.00	11741	0.48	90.57	84.38	99.59
	7	0.74	0.71	100.00	100.00	58175	1.54	90.57	84.38	99.59
`SortedList`	1	0.03	0.04	71.43	62.50	7	0.11	62.50	50.00	33.33
	2	0.04	0.07	100.00	100.00	36	0.11	80.00	74.14	52.81
	3	0.07	0.07	100.00	100.00	188	0.15	92.50	89.66	90.04
	4	0.22	0.20	100.00	100.00	1066	0.28	92.50	89.66	93.93
	5	0.53	0.48	100.00	100.00	7427	0.50	92.50	89.66	96.53
	6	1.94	1.77	100.00	100.00	73263	2.57	92.50	89.66	97.40
	7	22.68	21.13	100.00	100.00	1047608	37.91	92.50	89.66	97.40
`TreeMap`	1	0.02	0.02	57.14	63.33	6	0.06	14.41	14.89	5.46
	2	0.03	0.03	100.00	100.00	28	0.06	45.95	50.00	28.66
	3	0.07	0.04	100.00	100.00	96	0.09	63.96	73.40	61.09
	4	0.18	0.15	100.00	100.00	328	0.15	89.19	85.11	78.15
	5	0.38	0.31	100.00	100.00	1150	0.24	100.00	91.49	87.37
	6	0.94	0.61	100.00	100.00	3924	0.38	100.00	91.49	89.76
	7	3.28	1.75	100.00	100.00	12754	0.73	100.00	91.49	89.76
`HashSet`	1	0.01	0.01	57.89	69.23	4	0.04	51.92	50.00	29.91
	2	0.01	0.01	89.47	92.31	34	0.05	96.15	95.00	77.45
	3	0.06	0.05	89.47	92.31	212	0.09	100.00	100.00	90.57
	4	0.23	0.22	89.47	92.31	1170	0.19	100.00	100.00	90.98
	5	0.36	0.34	89.47	92.31	3638	0.27	100.00	100.00	91.39
	6	0.91	0.71	89.47	92.31	12932	0.62	100.00	100.00	91.80
	7	3.38	2.88	89.47	92.31	54844	1.55	100.00	100.00	92.21
`AVTree`	1	0.01	0.01	53.33	56.25	2	0.07	55.29	51.92	40.00
	2	0.05	0.03	90.00	87.50	86	0.14	75.29	78.85	60.00
	3	0.21	0.17	96.67	96.88	1702	0.78	88.23	84.61	75.12
	4	3.16	1.86	96.67	96.88	27734	8.36	94.12	92.31	91.21
	5	87.13	43.41	96.67	96.88	417878	134.51	94.12	92.31	93.65

Table 4: Korat's performance for test generation (with regular and dedicated generators), specification coverage (statement and branch), correctness checking, code coverage (statement and branch), and rate of mutant killing. All times are elapsed real times in seconds from the start of Korat to its completion.