Evaluating the “Small Scope Hypothesis”

Alexandr Andoni Dumitru Daniliuc Sarfraz Khurshid Darko Marinov
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139
{andoni,dumi,khurshid,marinov}@lcs.mit.edu

Abstract

The “small scope hypothesis” argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope. In object-oriented programs, a test input is constructed from objects of different classes; a test input is within a scope of s if at most s objects of any given class appear in it. If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope.

This paper evaluates the hypothesis for several implementations of data structures, including some from the Java Collections Framework. We measure how statement coverage, branch coverage, and rate of mutant killing vary with scope. For systematic input generation and correctness checking of Java programs, we use the Korat framework. This paper also presents the Ferastrau framework that we have developed for mutation testing of Java programs. The experimental results show that exhaustive testing within small scopes can achieve complete coverage and kill most of the mutants, even for intricate methods that manipulate complex data structures. The results also show that Korat can be used effectively to generate inputs and check correctness for these scopes.

1. Introduction

The “small scope hypothesis” argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope [13]. In object-oriented programs, a test input is constructed from objects of different classes; a test input is within a *scope* of s if at most s objects of any given class appear in it. If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope. This is one of the justifying principles of model checking [8].

Several case studies [14, 16] used the Alloy specification language [12] to develop software models and exhaustively check them for small scopes with the Alloy Analyzer. These case studies showed that the hypothesis holds for those software models, but they did not consider the actual implementations.

This paper evaluates the “small scope hypothesis” for several benchmark programs, including some data structure implementations from the Java Collections Framework [29]. Evaluating the hypothesis requires determining the scope up to which each program should be checked, i.e., the sufficient scope that gives significant confidence that the program has no bugs. We use code coverage and mutation testing criteria to determine the sufficient scope.

Code coverage is a common criterion for assessing the quality of a set of test inputs [5]. Measuring code coverage involves executing the program on each input and recording parts of the program (e.g., statements, branches, paths) that get executed. Statement (branch) coverage is then the ratio of the number of executed statements (branches) to the number of total statements (branches) in the program; *complete coverage* is the ratio of 100%.

Mutation testing is another criterion for assessing the quality of a set of test inputs [11, 26]. Mutation testing proceeds in two steps. In the first step, several *mutant*s are generated from the original (correct) program, by performing one or more syntactic modifications. These modifications are specified by *mutation operators*, e.g., replacing a variable with another variable (of a compatible type), say replacing `n.left` with `n.right`. For several languages, including Java, possible mutation operators are characterized in [2, 17–19, 27]. 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 than 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. Mutation testing frameworks were implemented for some languages, such as Mothra [19] for Fortran and Proteum [10] for C. We have implemented

Ferastrau (Section 4) for Java; to the best of our knowledge, the first framework for mutation testing of Java programs.

To perform exhaustive testing of Java programs, we use Korat [6] (Section 3), an automated framework for systematic input generation and correctness checking. Korat uses specification-based (or black-box) testing [5]. Given a formal specification for a method, Korat automatically generates all nonisomorphic test inputs (within a given small scope) that satisfy the method precondition. Korat then executes the method on each test input and uses the method postcondition as a test oracle to check the correctness of each output. For specifications, Korat currently uses the Java Modeling Language (JML) [20], and for checking correctness, Korat builds on the JML tool-set [7].

The experimental results show that systematic testing within small scopes can achieve complete coverage and kill almost all of the mutants, even for intricate methods that manipulate complex data structures. Furthermore, evaluating the "small scope hypothesis" is not simply about determining the actual value of the sufficient scope (be it 3 or 3000000) for each benchmark, but determining whether a framework that does systematic testing can be practically used for that scope. The experimental results also show that for all benchmarks and their sufficient scopes, Korat can generate all inputs and check correctness in less than an hour, often within a few seconds. These results provide evidence in support of the "small scope hypothesis". We believe that Korat and other frameworks that perform exhaustive checking within a small scope, such as JAlloy [15] and TestEra [24], are worth pursuing.

The main contributions of this paper are:

- Evaluation of the "small scope hypothesis" for several data structure implementations;
- Design and implementation of Ferastrau, a framework for mutation testing of Java programs;
- Evaluation of the Korat framework.

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. The example illustrates testing a method that manipulates linked data structures. Korat can be also used to test array-based data structures [6].

Consider the following Java code that declares a binary tree and its `remove` method:

```
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; // right child
    }

    boolean remove(Comparable info) { ... }
}
```

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 `remove` method.

The following JML annotations specify partial correctness for the example `remove` method:

```
class SearchTree {
    /*@ normal_behavior // 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 data is ordered
        if (!isOrdered(root)) return false;
        return true;
    }
}
```

The `normal_behavior` annotation specifies that if the precondition, annotated using `requires`, is satisfied at the beginning of the method, then the postcondition, annotated using `ensures`, is satisfied at the end of the method and the method returns without raising an exception. The method `repOk` (also known as `checkRep` [23]) is a Java predicate that checks the representation invariant of the corresponding data structure. For illustrative purposes, we put `repOk` in the precondition and postcondition; in practice, it is usually given as a class invariant, annotated using `invariant`, that is implicitly conjugated with the precondition and postcondition [20]. Good programming practice [23] suggests that implementations of abstract data types provide these predicates, as they are useful for checking correctness of the implementations.

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 and that the elements are in order, i.e., all elements in the left (right) subtree of a node are smaller (larger) than the element in that node. Appendix A shows the full code for `repOk`.

The method `contains` checks that the tree contains the given element. The JML keyword `\result` denotes the return value of the method. In this example, `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.

To test the `remove` method, Korat first generates valid inputs for the method. Each input is a pair of a tree and an

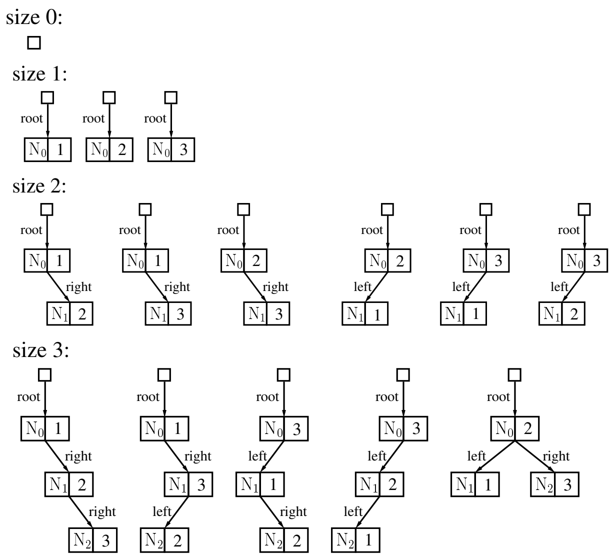

Figure 1. Trees generated for scope three.

element. The precondition defines valid inputs: the tree is valid, and the element is unconstrained. To limit the number of inputs, Korat uses a *finitization* (Section 3.1.1) description that specifies bounds on both the number of objects to be used to construct instances of the data structures and the values stored in the fields of these objects. Two trees are *isomorphic* if they have the same branching structure and isomorphic elements, irrespective of the identity of the actual objects in the trees.

Given a finitization, Korat generates all nonisomorphic input pairs, within the specified bounds, that satisfy the precondition. For example, in the scope three, i.e., using a maximum of three nodes and three elements, Korat generates 45 input pairs. These pairs are the Cartesian product of the 15 trees shown in Figure 1 and the three elements. For the `SearchTree` benchmark, we use Korat to generate inputs and check correctness of `remove` and `add` methods. As another example, in the scope seven, Korat generates 41300 input pairs for both these methods in less than ten seconds.

Korat uses the JML tool-set to translate method postconditions into runtime assertions [7]. After generating the inputs, Korat invokes the method instrumented with runtime 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, Korat takes less than two seconds to check both `remove` and `add` methods for all 41300 inputs.

We evaluate the "small scope hypothesis" by measuring how coverage and the rate of mutant killing vary with the scope. We use our Ferastrau framework for mutation testing. The "output" for `remove` consists of both its `boolean` return value and the post-state, i.e., the value of the receiver tree in the post-state. Figure 2 shows the variation for the `SearchTree` benchmark. Observe that a certain small scope is sufficient to achieve complete coverage and kill most of the mutants for this benchmark. Furthermore, Korat generates inputs and checks correctness for these scopes in less than 10 seconds.

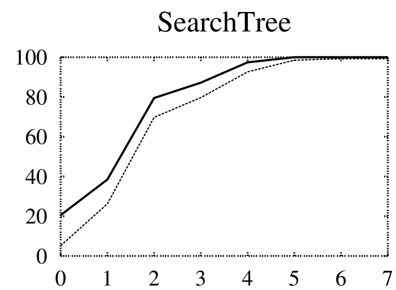

Figure 2. Variation of statement coverage (thick line) and rate of mutant killing (thin line) with scope.

3. Test generation and correctness checking

This section describes Korat [6], a framework for specification-based testing of Java programs. Korat automates both test-input generation and correctness checking. To check a method, Korat first systematically generates inputs that satisfy the method precondition. It then executes the method on these inputs and checks each output using the method postcondition as a test oracle.

3.1. Test input generation

The heart of Korat is a technique for generation of test inputs: given a Java predicate and a bound for its input, Korat automatically generates all nonisomorphic inputs for which the predicate returns `true`. Korat uses a *finitization* (described in Section 3.1.1) to bound the *state space* (Section 3.1.2) of predicate inputs. Korat uses backtracking (Section 3.1.3) to systematically explore the state space. Korat generates *candidate inputs* and checks their validity by invoking the predicate on them. Korat monitors accesses that the predicate makes to the fields of a candidate input. To monitor the accesses, Korat instruments the predicate and all the methods that the predicate transitively invokes. If the predicate returns without reading some fields of the candidate, the validity of the candidate must be independent of the values of those fields. Korat uses this observation to prune the search. Korat also uses an optimization that generates only nonisomorphic test inputs.

To generate inputs for a method, Korat first constructs a Java predicate corresponding to the method precondition. Next, Korat generates inputs for which the predicate returns `true`; each of these inputs corresponds to a valid test input

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

Figure 3. Finitization for the `remove` method.

for the method. For the `remove` method from Section 2, the corresponding predicate, `preRemove`, simply invokes `repOk` on the (implicit) `this` parameter.

3.1.1 Finitization

To generate a finite state space for the predicate's inputs, the search algorithm needs a finitization. Finitization is a set of bounds that limits the size of the inputs. Since the inputs can consist of objects from several classes, 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 some class domains.

In the spirit of using the implementation language familiar to programmers for specification and testing [3,4], Korat provides a `Finitization` class that allows finitizations to be given in Java. Korat automatically generates a finitization *skeleton* from the type declarations in the Java code [6]. Programmers can further specialize or generalize the skeleton.

Figure 3 shows two finitizations for the example `remove` method. For `finSearchTree_remove(s)`, Korat generates all inputs within 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: the fields `root`, `left`, and `right` can point to either `null` or a `Node` object; the `size` field ranges between `minSize` and `maxSize` (specified using the utility class `IntSet`); the `info` field and the `info` parameters are `Integer`s between `minInfo` and `maxInfo` (specified using the utility class `IntegerSet`). The Korat package provides several additional classes for easy construction of class domains and field domains.

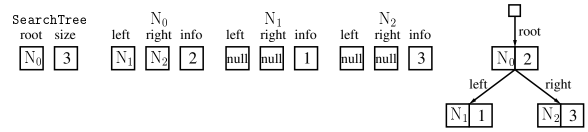

Figure 4. Candidate that is a valid `SearchTree`.

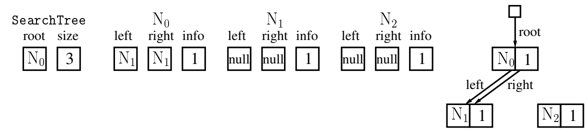

Figure 5. Candidate that is not a valid `SearchTree`.

3.1.2 State space

We continue with the `SearchTree` example to illustrate how Korat uses the finitization presented in Figure 3 to construct the state space of inputs to method `remove`. Consider the case when Korat is invoked for `finSearchTree_remove(3)`. Korat first allocates one `SearchTree` object and three `Node` objects. The three `Node` objects form the `Node` class domain. Korat then assigns a field domain and a unique identifier to each field. The identifier is the index into the *candidate vector*. In this example, the vector has 12 elements. The first 11 elements correspond to the fields of objects that form the input tree: the single `SearchTree` object has two fields (`root` and `size`) and the three `Node` objects have three fields each (`left`, `right`, and `info`). The last element represents the value of the method parameter `info`.

A *candidate* input is represented by a valuation of the candidate vector. The state space of inputs consists of all possible valuations of the candidate vector; each element of the vector represents a value from the corresponding field domain. In this example, the domain for fields `root`, `left`, and `right` has four elements (`null` and three `Node`s from the `Node` class domain), the domains for field `info` and parameter `info` have three elements, and the domain for field `size` has four elements. The state space, therefore, has $4*4*(4*4*3)^3*3 = 5308416 > 2^{22}$ potential candidates. For `scope`$= n$, the state space has $(n+1)^{2(n+1)} \cdot n^{n+1}$ potential candidates. Figure 4 shows an example candidate tree (from an input pair) that is a valid binary search tree with three nodes. Not all valuations represent valid binary search trees. Figure 5 shows an example candidate tree that is not a tree; `repOk` returns `false` for this candidate.

3.1.3 Search

To systematically explore the state space, Korat orders all the elements in every class domain and every field domain. The ordering in each field domain is consistent with the orderings in the class domains, and all the values that belong to the same class domain occur consecutively in the ordering of each field domain.

Each candidate input is a vector of *field domain indices* into the corresponding field domains. For our running example with `scope = 3`, assume that: the `Node` class domain is ordered as [N_0,N_1,N_2]; the field domain for fields `root`, `left`, and `right` is ordered as [`null`,N_0,N_1,N_2] (`null` by itself forms a class domain); the domain of the `size` field is ordered as `[0,1,2,3]`; and the domain of the field `info` and parameter `info` is ordered as [`Int(1)`,`Int(2)`,`Int(3)`]. According to this ordering, the candidate input that represents the candidate tree in Figure 4 (Figure 5) and value `Int(1)` for parameter `info` corresponds to the valuation `[1,3,2,3,1,0,0,0,0,0,2,0]` (`[1,3,2,2,0,0,0,0,0,0,0,0]`) for candidate vector.

The search starts with the candidate vector set to all zeros. For each candidate, Korat sets fields in the objects according to the values in the vector. Korat then invokes the predicate to check the validity of the current candidate. During the execution of the predicate, Korat monitors the fields that the predicate accesses. Specifically, Korat builds a *field-ordering*: a list of the field identifiers ordered by the first time the predicate accesses the corresponding field. As an illustration, consider the invocation of `preRemove` on the candidate tree shown in Figure 5 with value `Int(1)` for parameter `info`. In this case, `preRemove` accesses only the fields [`root`,N_0`.left`,N_0`.right`] (in that order) before returning `false`. Hence, the field-ordering that Korat builds is `[0,2,3]`.

After the predicate returns, Korat generates the next candidate vector backtracking on the fields accessed by the predicate. Korat first increments the field domain index for the field that is last in the field-ordering. If the domain index exceeds the domain size, Korat resets that index to zero, and increments the domain index of the previous field in the field-ordering, and so on. Continuing with our example, the next candidate takes the next value for N_0`.right`, which is N_2 by the above order, whereas the other fields do not change. This prunes from the search all $4^5 \cdot 3^4 = 82944$ candidate vectors of the form `[1,_,2,2,_,_,_,_,_,_,_,_]` that have the (partial) valuation: `root=`N_0, N_0`.left=`N_1, N_0`.right=`N_1. This pruning does not rule out any valid data structure because `preRemove` did not read the other fields, and it would have returned `false` irrespective of the values of those fields. If the predicate returns `true`, Korat outputs all (nonisomorphic) candidates that have the same values for the accessed fields as the current candidate. The search then backtracks to the next candidate.

Recall that Korat orders the values in the class and field domains. Additionally, each execution of the predicate on a candidate imposes an order on the fields in the field-ordering. Together, these orders induce a lexicographic order on the candidates. The Korat search algorithm generates inputs in the lexicographical order. Moreover, Korat avoids generating multiple candidates that are isomorphic to one another. Isomorphism between candidates partitions the state space into *isomorphism partitions*. For each isomorphism partition, Korat generates only the lexicographically least candidate in that partition.Conceptually, Korat avoids generating multiple candidates from the same isomorphism partition by incrementing field domain indices by more than one. More details on Korat can be found in [20].

	testing framework		
testing activity	JUnit	JML+JUnit	Korat
generating test inputs			√
generating test oracle		√	√
running tests	√	√	√

Table 1. Comparison of several testing frameworks for Java. Automated testing activities are indicated with '√'.

3.2. Checking correctness

To check a method, Korat first generates all valid inputs for the method using the process explained above. Korat then invokes the method on each of the inputs and checks each output with a *test oracle*. To check partial correctness of a method, a simple test oracle could just invoke `repOk` in the *post-state* (i.e., the state immediately after the method's invocation) 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.

The current Korat implementation uses the JML tool-set to automatically generate test oracles from method postconditions, as in the JML+JUnit framework [7]. The JML toolset translates JML postconditions into runtime Java assertions. If an execution of a method violates such an assertion, an exception is raised to indicate a violated postcondition. Test oracle catches these exceptions and reports correctness violations. These exceptions are different from the exceptions that the method specification allows, and Korat leverages on JML to check both normal and exceptional behavior of methods. More details on the JML tool-set and translation can be found in [20].

Korat can also use JML+JUnit to combine JML test oracles with 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. Korat additionally automates generation of test inputs, thus automating the entire testing process. Table 1 summarizes the comparison of these testing frameworks.

4. Mutation testing

This section presents design and implementation of Ferastrau, our framework for mutation testing of Java programs. *Mutation testing* is a criterion for assessing the quality of a set of test inputs [11, 26]. Mutation testing proceeds in two steps. In the first step, a set of *mutant*s is generated from the original program by performing one or more syntactic modifications. These modifications are performed by applying *mutation operators*. Section 4.1 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 than the original program, the test input is said to *kill* the mutant. Section 4.2 presents how Ferastrau executes mutants and compares the outputs.

4.1. Mutant generation

We have implemented mutant generation in Ferastrau by changing the Sun's `javac` compiler. Ferastrau performs a source-to-source translation: it first uses the compiler to parse each class of the original program into an abstract syntax tree; it then applies some mutation operators to the trees; and it finally uses the compiler to output the source of the mutants.

Ferastrau performs the following mutation operators:

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

The above operators modify only the code of methods, and not classes, e.g., by adding/removing a method or a field. It is easy to add new operators to Ferastrau to test different kind of mistakes. We believe that the current operators are representative for mistakes that programmers typically make in the benchmarks listed in Section 5.1, which is the focus of this paper. Moreover, some of the operators correspond to subtle mistakes that manifest only for non-trivial inputs, as the results in Section 5.3 show.

4.2. Mutant execution

After generating the mutants, Ferastrau uses a set of test inputs to perform mutation testing. These inputs can be provided manually or generated automatically. In our experiments, we 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 is correct, that it terminates (either normally or exceptionally) for all test inputs, and that it produces a deterministic output. The executions of mutants do not need then to check the (JML) postconditions.

For mutation testing of Java programs, several questions arise:

- How to compare outputs and therefore 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 of the mutants?
- How to handle exceptional termination of the original program and the mutants?

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

Ferastrau uses `equals` for comparing outputs, following Java convention of using `equals` for equality comparisons of objects. Recall that "output" for a method refers to both the return value and (the objects in) the post-state. Using `equals` allows comparisons based on abstract values. For example, two binary search trees that implement sets may be structurally different, but if they represent the same set, they would be considered equal according to the `equals` method. For most benchmarks, we have found their `equals` methods to suffice for comparing (non-exceptional) outputs; in general, another method may be used.

For a class, say `C`, `equals` is often implemented as:

```
public boolean equals(Object o) {
    if (!(o instanceof C)) return false;
    C c = (C)o;
    ...
}
```

Note that using `equals` for comparing outputs requires that mutant classes have the same name as the corresponding original classes, and Ferastrau generates such mutant classes. An alternative would be to rename the classes during the mutant generation, say `C` to `C_mutant`. However, that would require the users of Ferastrau to provide special methods for comparing outputs.

Ferastrau executes the original program and the mutants in a single run. This introduces different classes with the same name in a single Java Virtual Machine (JVM). Ferastrau achieves this by using a different `ClassLoader` [29] to load in the classfiles of each mutant. To compare objects between these loaders and the original program (which is in the default `ClassLoader`), Ferastrau uses serialization through a buffer in memory. An alternative to the single run would be to first execute the original program for all inputs and serialize all outputs, and then execute each mutant. However, that would require storing all outputs, which can produce very large files, especially for inputs exhaustively generated by Korat.

Ferastrau handles non-termination of mutants by setting a time limit for execution. For each input, a mutant is run in a separate thread. If the mutant runs longer than it is allowed, the corresponding thread is stopped. Ferastrau sets the time limit 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.

Ferastrau assumes that the original program terminates for all test inputs, either normally or exceptionally. Note that these exceptions are allowed by the specification, and they are not errors. Similarly, the mutants can terminate either normally or exceptionally. Ferastrau catches all the exceptions that these executions raises, and it takes them into account when comparing outputs.

Ferastrau catches all exceptions (or, in terms of Java, all `Throwable` objects), and thus all the errors that the mutant executions may raise. This handles the situations when the mutant runs out of stack or heap memory and JVM raises `StackOverflowError` or `OutOfMemoryError`. Although the JVM specification [22] does not precisely specify the behavior of JVMs when `OutOfMemoryError` is raised, we found several Sun's Java 2 SDK1.3.x JVMs to be able to continue execution after garbage collection of the objects allocated by the erroneous mutant.

Ferastrau uses the following criteria to kill a mutant for some test input:

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

5. Experimental results

This section presents the experiments that evaluate the "small scope hypothesis" and the Korat framework. We first present Korat's performance results for test input generation and checking method correctness. We then present how the coverage and the rate of mutant killing vary with the scope. We performed all timed experiments on a Linux machine with a Pentium 4 1.8GHz processor using Sun's Java 2 SDK1.3.1 JVM.

5.1. Benchmarks and methods

Table 2 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 "tested" methods. These methods implement the standard operations on their corresponding data structures [9]. Executing these methods also tests some "helper" methods because they are invoked either from the code for "tested" methods or while checking the correctness of those methods (i.e., from postconditions).

`SearchTree` is presented in Section 2. `DisjSet` is an array-based implementation of the fast union-find data structure [9]; 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 [9].

`LinkedList` is the implementation of linked lists in the Java Collections Framework, a part of the standard Java libraries [29]. This implementation uses doubly-linked, circular lists The elements in `LinkedList` are arbitrary objects from a given set. `SortedList` is structurally identical to `LinkedList`, but the elements in `SortedList` are (sorted) `Integer`s. This benchmark is similar to the examples used in some shape analyses [21,25]. `TreeMap` implements the `Map` interface using red-black trees [9]. `HashSet` implements the `Set` interface, backed by a hash table [9].

`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. Each node in an intentional name has an `attribute`, a `value`, and a set of child nodes. Attributes and values are arbitrary `String`s (except that '`*`' matches all other values). The original implementation of INS had errors [24]; in these experiments we use the corrected version.

5.2. Test generation and correctness checking

Table 3 shows Korat's performance for test generation and correctness checking for some scopes. Appendix B presents the results also for other scopes. For each benchmark, all size parameters and maximum elements are set to the scope value. For each benchmark, the tabulated scopes are sufficient to achieve the maximum coverage and kill almost all the mutants. We also tabulate the time Korat takes to generate all valid test inputs, the number of inputs, and the time Korat takes 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.

Number of inputs that is generated is the sum of numbers of inputs for all "tested" methods. Similarly, the generation and checking times are sums of times for all "tested" methods. We use Korat to separately generate inputs for each method. However, when two methods have the same precondition (e.g., an `add` and a `remove` that only require their input data structure to be valid), we could reuse the inputs and thus even reduce the generation time. The postconditions for all methods specify typical partial correct-

benchmark	"tested" methods	some "helper" methods	# ncnb lines	# branches	# mutants
SearchTree	add, remove	contains	85	20	271
DisjSet	union, find	compressPath	29	8	284
HeapArray	insert, extractMax	heapifyUp, heapifyDown	51	9	274
BinomialHeap	insert, extractMin union, delete	contains, decrease merge, findMin	182	33	295
FibonacciHeap	insert, extractMin union, delete	contains, decrease cascadingCut, cut, consolidate	171	31	302
LinkedList	add, remove, reverse	contains, ListIterator.next	102	16	253
SortedList	insert, remove sort, merge	contains	176	29	253
TreeMap	put, remove	get, containsKey fixAfterInsertion fixAfterDeletion rotateLeft, rotateRight	230	47	294
HashSet	add, remove	contains, HashMap.containsKey HashMap.put, HashMap.remove HashMap.rehash	113	20	285
AVTree	lookup	extract	199	26	203

Table 2. Benchmarks and tested 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 "tested" methods, thereby also testing "helper" methods. For each benchmark, we tabulate the number of non-comment non-blank lines of source code in those methods, the number of branches, and the number of mutants generated by Ferastrau.

benchmark	scope	generation [sec]	# inputs	checking [sec]	statement cov. [%]	branch cov. [%]	mutants killed [%]
SearchTree	6	1.38	8772	0.43	100.00	100.00	99.26
DisjSet	4	0.31	18280	0.44	100.00	100.00	95.77
HeapArray	6	1.02	118251	1.94	100.00	100.00	95.98
BinomialHeap	7	232.33	2577984	67.05	100.00	100.00	95.93
FibonacciHeap	5	1258.95	941058	26.45	100.00	100.00	89.07
LinkedList	3	0.01	169	0.10	90.57	84.38	99.20
SortedList	6	2.55	73263	2.59	92.50	89.66	96.44
TreeMap	6	1.00	3924	0.39	100.00	91.49	89.11
HashSet	5	0.43	3638	0.29	100.00	100.00	92.28
AVTree	5	84.51	417878	148.05	94.12	92.31	93.10

Table 3. Korat's performance for test generation and correctness checking; also, statement and branch coverage and rate of mutant killing. All times are elapsed real times in seconds from the start of Korat to its completion. For all benchmarks and their sufficient scopes, Korat takes less than 0.5 hour to generate all inputs and check correctness.

ness properties; they require resulting data structures to be valid and either to contain or not contain the input elements, depending on the method.

For these scopes, the size of the search space is between 2^{25} and 2^{100}. The actual size of search spaces for several data structures can be found in [6]; for some scopes in those experiments, as well as for some scopes in Appendix B, Korat explores search spaces with size over 2^{200}. In all cases, Korat completes in less than 0.5 hour, often in just a few seconds.

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 structures. Isomorphism optimization further reduces the number of considered structures, but it mainly reduces the number of valid structures. As shown in [6], Korat generates exactly the number of nonisomorphic structures given in the On-Line Encyclopedia of Integer Sequences [28].

5.3. Coverage and mutant testing

Figure 6 shows graphs that relate scope with the statement coverage and the rate of mutant killing. The coverage is measured for all "tested" and "helper" methods, since they are all executed. For most benchmarks, Korat generates inputs that achieve complete coverage, both for statements and branches. For other benchmarks, the coverage is not complete because no input for "tested" methods could

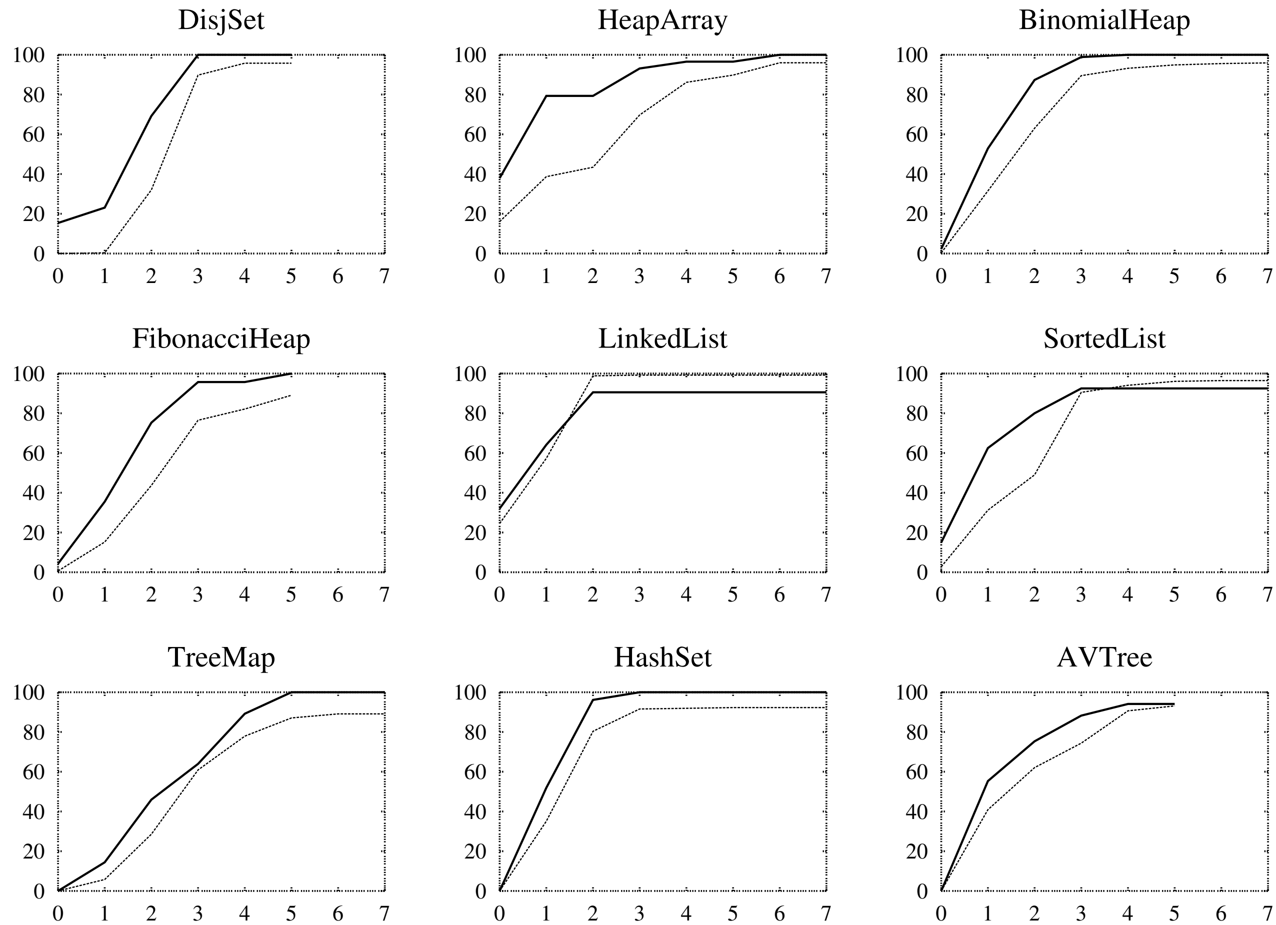

Figure 6. Variation of statement coverage (thick line) and rate of mutant killing (thin line) with scope. For *all* benchmarks, Korat generates inputs that achieve the maximum coverage that is possible without directly generating inputs for "helper" methods.

trigger some exceptional behavior of "helper" methods.

For example, the ("tested") `reverse` method for lists creates a `ListIterator` and invokes some ("helper") methods on it, but in a way that precludes raising certain exceptions, such as `ConcurrentModificationException` or `NoSuchElementException`. In terms of JML specifications, the "tested" methods typically invoke "helper" methods in pre-states that satisfy the precondition for `normal_behavior`, and not `exceptional_behavior`.

For mutant testing, we use Ferastrau to generate between 200 and 300 mutants for each benchmark. We instruct Ferastrau to mutate the "tested" methods and their "helper" methods, but not "helper" methods that are invoked only from specifications. For most benchmarks, Korat generates inputs that kill over 90% of the mutants. Our manual inspection of the surviving mutants indicates that most of them are semantically equivalent to the original programs and thus no input could kill them; due to the complexity of the benchmark methods, we were not able to definitely establish the equivalence for all mutants that are not killed.

Notice that for some of the benchmarks the rate of mutant killing increases with scope even after achieving complete coverage. This can be expected because complete statement and branch coverage (or for that matter, any coverage criteria) do not guarantee absence of bugs [5]. 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 body is executed zero or one times, the original program and the mutant have the same behavior. For trees that have up to four nodes, the loop cannot execute more than once; these trees also happen to achieve complete coverage. However, executing the method for a tree that has five nodes can execute the loop twice in the original program, thus making the mutant loop infinitely. (Recall that Ferastrau detects mutants that loop infinitely.)

Because of the above, we take as sufficient the scope for which almost all mutants are killed, and not the scope that just achieves complete coverage. For all benchmarks and their respective sufficient scopes, Korat can generate all inputs and check correctness using these inputs in less than an hour, often within a few seconds. Korat can therefore be used effectively for systematic testing of these benchmarks.

6. Conclusions

The "small scope hypothesis" argues that a high proportion of bugs can be found by testing the program for all test inputs within some small scope. In object-oriented programs, a test input is constructed from objects of different classes; a test input is within a scope of s if at most s objects of any given class appear in it. If the hypothesis holds, it follows that it is more effective to do systematic testing within a small scope than to generate fewer test inputs of a larger scope.

This paper evaluated the hypothesis for several implementations of data structures, including some from the Java Collections Framework. We measured how statement coverage, branch coverage, and rate of mutant killing vary with scope. To perform exhaustive testing, we used Korat, an automated framework for systematic input generation and correctness checking of Java programs. This paper also presented the Ferastrau framework that we developed for mutation testing of Java programs. The experimental results show that exhaustive testing within small scopes can achieve complete coverage and kill almost all of the mutants, even for intricate methods that manipulate complex data structures. The results also show that Korat can be used effectively to generate inputs and check correctness for these scopes. We believe that frameworks that perform exhaustive checking within a small scope, such as Korat, are worth pursuing.

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] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 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)*, July 2002.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] D. Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. Available online: http://sdg.lcs.mit.edu/alloy/book.pdf.

[13] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7), July 1996.

[14] D. Jackson and K. Sullivan. COM revisited: Tool-assisted modeling of an architectural framework. In *Proc. 8th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, San Diego, CA, 2000.

[15] 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.

[16] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.

[17] S.-W. Kim, J. Clark, and J. McDermid. The rigorous generation of Java mutation operators using HAZOP. In *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, Dec. 1999.

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

[19] 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.

[20] 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).

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

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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`.

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

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; // right child
    }

    /*@ normal_behavior // 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 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);
            }
        }
        // checks that size is consistent
        if (visited.size() != size) return false;
        return true;
    }

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

    private boolean isOrdered(Node n, Comparable min, Comparable max) {
        if (n == null) return true;
        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

benchmark	scope	generation [sec]	# inputs	checking [sec]	statement cov. [%]	branch cov. [%]	mutants killed [%]
SearchTree	1	0.03	4	0.05	38.46	40.00	26.19
	2	0.04	20	0.06	79.49	87.50	69.74
	3	0.06	90	0.06	87.18	92.50	79.70
	4	0.17	408	0.13	97.44	97.50	92.61
	5	0.38	1880	0.23	100.00	100.00	98.52
	6	1.38	8772	0.43	100.00	100.00	99.26
	7	8.96	41300	1.24	100.00	100.00	99.26
DisjSet	1	0.01	4	0.04	23.08	25.00	0.35
	2	0.01	30	0.04	69.23	68.75	32.04
	3	0.04	456	0.08	100.00	100.00	89.78
	4	0.31	18280	0.44	100.00	100.00	95.77
	5	14.24	1246380	20.98	100.00	100.00	95.77
HeapArray	1	0.01	16	0.04	79.31	66.67	38.68
	2	0.01	75	0.05	79.31	66.67	43.43
	3	0.01	396	0.08	93.10	83.33	69.70
	4	0.08	2240	0.16	96.55	88.89	86.13
	5	0.23	15352	0.38	96.55	94.44	89.78
	6	1.02	118251	1.94	100.00	100.00	95.98
	7	9.51	1175620	18.09	100.00	100.00	95.98
BinomialHeap	1	0.01	12	0.06	52.87	57.58	31.52
	2	0.01	54	0.07	87.36	84.85	63.05
	3	0.13	336	0.12	98.85	96.97	89.49
	4	0.45	1800	0.23	100.00	98.48	93.22
	5	2.02	16848	0.63	100.00	100.00	94.91
	6	13.90	159642	4.07	100.00	100.00	95.59
	7	232.33	2577984	67.05	100.00	100.00	95.93
FibonacciHeap	1	0.01	12	0.06	35.48	43.55	15.23
	2	0.08	108	0.08	75.27	80.64	43.70
	3	0.79	1632	0.23	95.70	98.39	76.49
	4	20.07	34650	1.14	95.70	98.39	82.11
	5	1258.95	941058	26.45	100.00	100.00	89.07
LinkedList	1	0.01	15	0.07	64.15	68.75	57.31
	2	0.01	50	0.07	90.57	84.38	98.81
	3	0.01	169	0.10	90.57	84.38	99.20
	4	0.08	627	0.14	90.57	84.38	99.20
	5	0.18	2584	0.24	90.57	84.38	99.20
	6	0.36	11741	0.49	90.57	84.38	99.20
	7	1.00	58175	1.60	90.57	84.38	99.20
SortedList	1	0.02	7	0.10	62.50	50.00	31.22
	2	0.02	36	0.11	80.00	74.14	49.01
	3	0.07	188	0.14	92.50	89.66	90.51
	4	0.22	1066	0.27	92.50	89.66	94.07
	5	0.56	7427	0.50	92.50	89.66	96.04
	6	2.55	73263	2.59	92.50	89.66	96.44
	7	33.12	1047608	38.16	92.50	89.66	96.44
TreeMap	1	0.02	6	0.06	14.41	14.89	5.78
	2	0.02	28	0.06	45.95	50.00	28.57
	3	0.08	96	0.09	63.96	73.40	60.88
	4	0.18	328	0.14	89.19	85.11	77.89
	5	0.37	1150	0.24	100.00	91.49	87.07
	6	1.00	3924	0.39	100.00	91.49	89.11
	7	3.33	12754	0.78	100.00	91.49	89.11
HashSet	1	0.01	4	0.03	51.92	50.00	35.08
	2	0.01	34	0.04	96.15	95.00	80.35
	3	0.07	212	0.08	100.00	100.00	91.57
	4	0.25	1170	0.19	100.00	100.00	91.92
	5	0.43	3638	0.29	100.00	100.00	92.28
	6	1.15	12932	0.59	100.00	100.00	92.28
	7	5.08	54844	1.96	100.00	100.00	92.28
AVTree	1	0.01	2	0.06	55.29	51.92	40.88
	2	0.04	86	0.13	75.29	78.85	62.06
	3	0.20	1702	0.75	88.23	84.61	74.38
	4	2.69	27734	8.32	94.12	92.31	90.64
	5	84.51	417878	148.05	94.12	92.31	93.10

Table 4. Korat's performance for test generation and correctness checking; also, variation of statement and branch coverage and rate of mutant killing with scope. All times are elapsed real times in seconds from the start of Korat to its completion. For all benchmarks and their sufficient scopes, Korat takes less than 0.5 hour to generate all inputs and check correctness.