# Korat: A Tool for Generating Structurally Complex Test Inputs

Aleksandar Milićević Saša Misailović
University of Belgrade
Belgrade, Serbia

Darko Marinov
University of Illinois
Urbana-Champaign, IL

Sarfraz Khurshid
University of Texas
Austin, TX

## Abstract

*This paper describes the Korat tool for constraint-based generation of structurally complex test inputs for Java programs. Korat takes (1) an imperative predicate that specifies the desired structural integrity constraints and (2) a finitization that bounds the desired test input size. Korat generates all inputs (within the bounds) for which the predicate returns true. To do so, Korat performs a systematic search of the predicate's input space. The inputs that Korat generates enable bounded-exhaustive testing for programs ranging from library classes to stand-alone applications.*

## 1. Introduction

Testing is the dominant method for finding software errors in practice. As such, testing is critical to the production of high-quality code. A key requirement for successfully testing any software system is obtaining good test inputs. Manual generation of test inputs is laborious and typically produces inputs that exercise only a small subset of the functionality of the software. The alternative is to automate generation of test inputs; such automation can significantly help the developers produce and maintain reliable code.

We have developed the Korat tool for automated testing of Java programs [1, 6]. Korat focuses on programs that have *structurally complex* inputs: the inputs are structural—e.g., represented with linked data structures—and must satisfy complex properties that relate parts of the structure—e.g., invariants for linked data structures. Almost all modern software systems manipulate structurally complex data. For example, Java programs operate on a heap that consists of linked objects; each heap configuration must satisfy the consistency properties of the data structures in the heap. As another example, Web services manipulate XML documents; each service operates only on documents that satisfy certain syntactic and semantic properties.

To illustrate the testing of code that takes structurally complex inputs, consider the testing of an operation on some data structure that implements an abstract data type. For example, consider a Java method that removes an element from a tree that implements a set. Each such operation should preserve the invariant of the data structure. In this example, the invariant is that the object graph that represents the tree is indeed a tree and not an arbitrary graph with cycles or sharing. To test the remove method, we need to execute it on several input object graphs. This method has an implicit precondition: the input must be a tree. Thus, we do not want to test the remove method for arbitrary object graphs—it may very well loop infinitely or return an incorrect output. To test remove, we need to generate object graphs that satisfy the tree invariant.

As another example, consider the testing of a program that processes XML documents. The input to the program is simply a sequence of characters. If the sequence does not satisfy a syntactic or semantic property of the XML documents, the program only reports an error. It is important to generate test inputs that check this error-reporting behavior. Additionally, we need to check the actual processing, which requires generating test inputs that satisfy the properties of the kind of XML documents that the program processes.

The Korat tool implements a solver for *imperative predicates* that express structural invariants in Java code. The solver takes an imperative predicate and additionally a *finitization* that bounds the size of the structures that are inputs to the predicate. The solver systematically searches the bounded structure space, effectively pruning large portions of the space, and outputs all nonisomorphic structures (within the given bounds) for which the predicate returns true. These structures can form a *bounded-exhaustive* test suite for testing the code on all inputs within the given small bound. We used such test suites previously [1,6], and the results show that Korat can efficiently generate test suites that achieve high statement, branch, and mutation coverage [7].

This paper summarizes the technique and presents a publicly available version of Korat. Korat allows the structures to be printed, serialized to disk, or viewed graphically as object graphs. Korat currently leverages the Alloy Analyzer's visualization facility [4] to provide a customizable graphical display. Korat is available for download from `http://mir.cs.uiuc.edu/korat`.

```
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
        int info; // data
    }

    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 structure and invariant**

## 2. Example

This section illustrates Korat's generation. We use binary search trees as a running example. Figure 1 shows Java code that defines a binary search tree. The method `repOk` is a Java predicate that checks the *representation invariant* [5] of `SearchTree`. First, `repOk` checks if the tree is empty. If not, `repOk` checks that there are no undirected cycles along the `left` and `right` fields, that the number of nodes reachable from `root` is the same as the value of `size`, and that all elements in the left (right) subtree of a node are smaller (larger) than the element in that node.

Korat can generate valid binary search trees. To limit the number of generated structures, Korat uses a *finitization* (Section 3.1) that bounds the number of objects in the data structures and the field values of these objects. For trees, finitization gives the maximum number of nodes and the possible values in nodes. Following Alloy's terminology for bounds [4], we say that a tree is in *scope s* if it has at most *s* nodes and *s* values. Two trees are *isomorphic* if they have the same shape (branching structure) and (primitive) elements, regardless of the identity of the actual nodes in the trees.

Given a finitization and a value for scope, Korat generates all non-isomorphic structures that satisfy the class invariant. For example, in scope three, Korat generates the 15 trees shown in Figure 2 in less than one second.

It is practical to use Korat to generate inputs that give high code and mutation coverage [7]. To illustrate, consider the method `remove` that removes a given element from a given tree. Figure 3 shows how statement coverage and the rate of mutant killing [7] vary with the scope for this method. Scope five is sufficient to achieve complete coverage, and scope six is sufficient to kill all non-equivalent mutants. Generating inputs and checking correctness for these scopes using Korat takes just a few seconds.

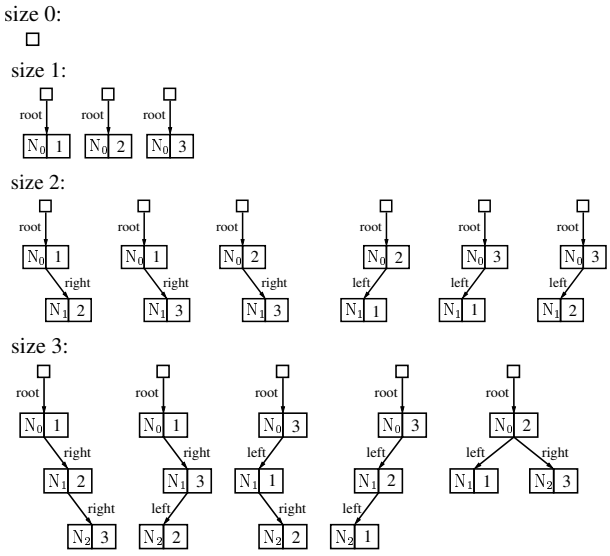


**Figure 2. Trees generated for scope three**

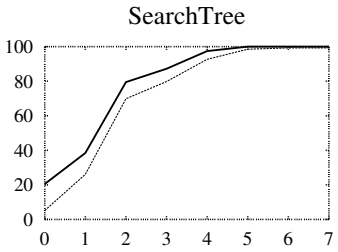


**Figure 3. Variation of statement coverage (thick line) and rate of mutant killing (thin line) with scope**

## 3. Korat

Given a Java predicate and a bound on the predicate's input, Korat generates all non-isomorphic inputs that are *valid*, i.e., inputs for which the predicate returns `true` [1,6]. Korat uses a *finitization* (Section 3.1) to bound the *state space* (Section 3.2) of predicate inputs. Korat generates *candidate* structures and invokes the predicate on them. Korat uses backtracking (Section 3.3) to systematically explore this state space. The Korat implementation we present uses bytecode instrumentation (Section 3.4) and allows generated structures to be printed, serialized to disk, or visualized graphically (Section 3.5).

Korat uses two optimizations for efficient generation. First, it prunes the search based on the fields that the predicate accesses. To monitor the accesses, Korat instruments all classes that appear in finitizations. Second, it generates only non-isomorphic candidates. These optimizations speed up the search without compromising its correctness.

```
IFinitization finSearchTree(int numNode,
    int minSize, int maxSize, int minInfo, int maxInfo) {
  IFinitization f = new Finitization(SearchTree.class);
  IObjSet nodes = f.createObjSet(Node.class, numNode);
  nodes.setNullAllowed(true);
  f.set("root", nodes);
  f.set("size", f.createIntSet(minSize, maxSize));
  f.set("Node.left", nodes);
  f.set("Node.right", nodes);
  f.set("Node.info", f.createIntSet(minInfo, maxInfo));
  return f;
}
IFinitization finSearchTree(int scope) {
  return finSearchTree(scope, 0, scope, 1, scope);
}
```

**Figure 4. Two finitizations for `SearchTree`**

Each candidate that Korat generates is an object graph with one root. Executing the same Java program from two isomorphic states should not lead to observable difference in the executions. Thus, we define structure isomorphism based on object identity: two candidates are isomorphic iff the object graphs reachable from the root are isomorphic.

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 so explicitly; instead, it avoids generating isomorphic valid inputs by not even considering isomorphic candidates.

In summary, Korat generates all non-isomorphic valid inputs within given bounds, and its search has the following properties:

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

### 3.1. Finitization

To generate a finite state space for predicate's inputs, Korat uses a finitization that limits the input size. The inputs can consist of objects from several classes, and the finitization specifies the number of objects for each class. 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*, a union of several class domains.

Korat provides a `Finitization` class that allows finitizations to be written in Java. Figure 4 shows two finitizations for `SearchTree`; invoking `finSearchTree(s)` creates a finitization for scope `s`. The `createObjSet` method specifies that the input contains at most `numNode` objects from the class `Node`. The `set` method specifies a field domain for each field.

### 3.2. State space

Korat uses a finitization to construct a state space of predicate inputs. For example, consider the finitization `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 (Section 3.3) uses indexes into these domains.

Korat next assigns a field domain to each field. 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` plus three `Node` objects. The `null` value and each primitive value (of type `int`, `boolean` etc.) 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. For 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 $4 \cdot 4 \cdot (4 \cdot 4 \cdot 3)^3 > 2^{20}$ 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 exploration 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.

### 3.3. Search

The search starts with the state set to all zeros. For each state, Korat first creates the corresponding candidate. Korat then executes the predicate on the candidate to check its validity. 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.

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 by 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 becomes empty. 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.

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. Thus, 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 more details, see elsewhere [1, 6].

### 3.4. Instrumentation

To monitor field accesses during `repOk`'s executions, Korat performs bytecode instrumentation. Korat uses the Bytecode Engineering Library [3] and the Javassist framework [2].

Korat instruments all classes whose objects appear in finitizations. For each class, Korat adds a special constructor. For each field of those classes, Korat adds an identifier field and special getter and setter methods. In each method of those classes (including `repOk`), Korat replaces each field access with an invocation of the corresponding getter or setter method. Arrays are similarly instrumented, essentially treating each array element as a field.

To monitor the field accesses and build a field-ordering, Korat uses an approach similar to the observer design pattern. Korat uses the special constructors to conceptually initialize all objects in a finitization with an observer. While building state space, Korat initializes each of the identifier fields to a unique index into the candidate vector. Special getter and setter methods first notify the observer of the field access using the field's identifier and then perform the field access (return the field's value or assign to the field).

### 3.5. Visualization

Korat can graphically show the structures it generates. The visualization in Korat was inspired by Alloy [4], and our current Korat implementation uses the Alloy Analyzer's visualization facility [4], which provides a fully customizable display that allows users to specify desired views on the underlying structures. Korat automatically translates object graphs into the Alloy representation. The visualization assists users in writing correct `repOk` methods and also in understanding any faults revealed in the code tested with the Korat-generated inputs.

To illustrate, the command:

```
java korat.Korat -class SearchTree -visualize -params 3 0 3 1 3
```

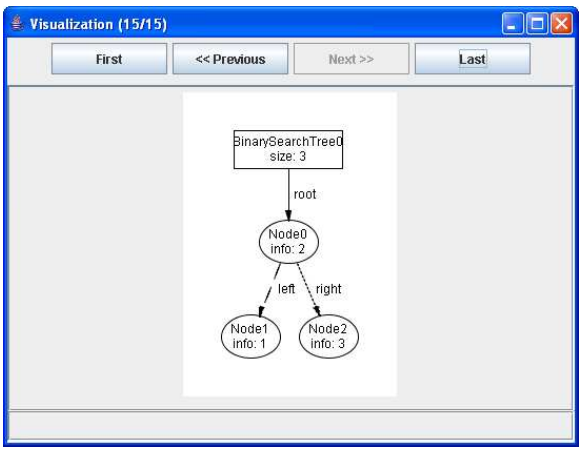


**Figure 5. Example `SearchTree` visualization**

executes Korat (1) to generate all binary search trees with up to 3 nodes with `info` values ranging from 1 to 3 and (2) to display graphically the generated structures. Figure 5 shows an example visualization window. The `First`, `Previous`, `Next`, and `Last` buttons allow scrolling through the list of generated structures.

## Acknowledgments

We thank Brett Daniel for extensive comments on an earlier draft of this paper. We thank Jesus DeLaTorre and ChoongHwan Lee for comments on the Korat tool. This material is based upon work partially supported by the NSF under Grant Nos. 0438967, 0613665, and 0615372. We also acknowledge support from Microsoft Research.

## References

[1] 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.

[2] S. Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct. 1998.

[3] M. Dahm. Byte code engineering library. `http://bcel.sourceforge.net/`.

[4] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

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

[6] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[7] 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.