ACM International Symposium on Software Testing and Analysis (ISSTA), July 2002. Winner of an ***ACM SIGSOFT Distinguished Paper Award.***

Korat: Automated Testing Based on Java Predicates

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{chandra,khurshid,marinov}@lcs.mit.edu

ABSTRACT

This paper presents Korat, a novel framework for automated testing of Java programs. Given a formal specification for a method, Korat uses the method precondition to automatically generate all (nonisomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output.

To generate test cases for a method, Korat constructs a Java predicate (i.e., a method that returns a boolean) from the method's precondition. The heart of Korat is a technique for automatic test case generation: given a predicate and a bound on the size of its inputs, Korat generates all (nonisomorphic) inputs for which the predicate returns true. Korat exhaustively explores the bounded input space of the predicate but does so efficiently by monitoring the predicate's executions and pruning large portions of the search space.

This paper illustrates the use of Korat for testing several data structures, including some from the Java Collections Framework. The experimental results show that it is feasible to generate test cases from Java predicates, even when the search space for inputs is very large. This paper also compares Korat with a testing framework based on declarative specifications. Contrary to our initial expectation, the experiments show that Korat generates test cases much faster than the declarative framework.

1. INTRODUCTION

Manual software testing, in general, and test data generation, in particular, are labor-intensive processes. Automated testing can significantly reduce the cost of software development and maintenance [4]. This paper presents Korat, a novel framework for automated testing of Java programs. Korat uses specification-based testing [5, 13, 15, 25]. Given a formal specification for a method, Korat uses the method precondition to automatically generate all nonisomorphic test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output.

To generate test cases for a method, Korat constructs a Java predicate (i.e., a method that returns a boolean) from the method's precondition. One of the key contributions of Korat is a technique for automatic test case generation: given a predicate, and a bound on the size of its inputs, Korat generates all nonisomorphic inputs for which the predicate returns `true`. Korat uses backtracking to systematically explore the bounded input space of the predicate. Korat generates *candidate* inputs and checks their validity by invoking the predicate on them. Korat monitors accesses that the predicate makes to all the fields of the candidate input. If the predicate returns without reading some fields of the candidate, then the validity of the candidate must be independent of the values of those fields—Korat uses this observation to prune large portions of the search space. Korat also uses an optimization to generate only nonisomorphic test cases. (Section 3.4 gives a precise definition of nonisomorphism.) This optimization reduces the search time without compromising the exhaustive nature of the search.

Korat lets programmers write specifications in any language as long as the specifications can be automatically translated into Java predicates. We have implemented a prototype of Korat that uses the Java Modeling Language (JML) [20] for specifications. Programmers can use JML to write method preconditions and postconditions, as well as class invariants. JML uses Java syntax and semantics for expressions, and contains some extensions such as quantifiers. A large subset of JML can be automatically translated into Java predicates. Programmers can thus use Korat without having to learn a specification language much different than Java. Moreover, since JML specifications can call Java methods, programmers can use the full expressiveness of the Java language to write specifications.

To see an illustration of the use of Korat, consider a method that removes the minimum element from a balanced binary tree. The (implicit) precondition for this method requires the input to satisfy its class invariant: the input must be a binary tree and the tree must be balanced. Korat uses the code that checks the class invariant as the predicate for generating all nonisomorphic balanced binary trees bounded by a given size. Good programming practice [21] suggests that implementations of abstract data types provide predicates (known as the `repOk` or `checkRep` methods) that check class invariants—Korat then generates test cases almost for free. Korat invokes the method on each of the generated trees and checks the postcondition in each case. If a method postcondition is not (explicitly) specified, Korat can still be used to test partial correctness of the method. In the binary tree example, Korat can be used to check the class invariant at the end of the remove method, to see that the tree remains a balanced binary tree after removing the minimum element from it.

ISSTA'02, July 22-24, 2002, Rome, Italy.

```
import java.util.*;
class BinaryTree {
    private Node root; // root node
    private int size;  // number of nodes in the tree
    static class Node {
        private Node left;  // left child
        private Node right; // right child

    }
    public  boolean repOk() {
        // checks that empty tree has size zero
        if (root == null) return size == 0;
        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 tree has no cycle
                if (!visited.add(current.left))
                    return false;
                workList.add(current.left);
            }
            if (current.right != null) {
                // checks that 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;
    }
}
```

Figure 1: `BinaryTree` example

We have used Korat to test several data structures, including some from the Java Collections Framework. The experimental results show that it is feasible to generate test cases from Java predicates, even when the search space for inputs is very large. In particular, our experiments indicate that it is practical to generate inputs to achieve complete statement coverage, even for intricate methods that manipulate complex data structures. This paper also compares Korat with the Alloy Analyzer [16], which can be used to generate test cases [22] from declarative predicates. Contrary to our initial expectation, the experiments show that Korat generates test cases much faster than the Alloy Analyzer.

The rest of this paper is organized as follows. Section 2 illustrates the use of Korat on two examples. Section 3 presents the algorithm that Korat uses to explore the search space. Section 4 describes how Korat checks method correctness. Section 5 presents the experimental results. Section 6 reviews related work, and Section 7 concludes.

2. EXAMPLES

This section presents two examples to illustrate how programmers can use Korat to test their programs. These examples, a binary tree data structure and a heap[1] data structure, illustrate methods that manipulate linked data structures and array-based data structures, respectively.

2.1 Binary tree

This section illustrates the generation and testing of linked data structures using simple binary trees. The Java code in Figure 1 declares a binary tree and defines its `repOk` method, i.e., a Java predicate that checks the representation invariant (or class invariant) of the corresponding data structure [21]. In this case, `repOk` checks if the input is a tree with the correct `size`.

[1]The term "heap" refers to the data structure (priority queues) and not to the garbage-collected memory.

```
public static Finitization finBinaryTree(int NUM_Node) {
  Finitization f = new Finitization(BinaryTree.class);
  ObjSet nodes = f.createObjects("Node", NUM_Node);
                             // #Node = NUM_Node
  nodes.add(null);
  f.set("root", nodes);       // root in null + Node
  f.set("size", NUM_Node);    // size = NUM_Node
  f.set("Node.left", nodes);  // Node.left in null + Node
  f.set("Node.right", nodes); // Node.right in null+ Node
  return f;
}
```

Figure 2: Finitization description for the `BinaryTree` example

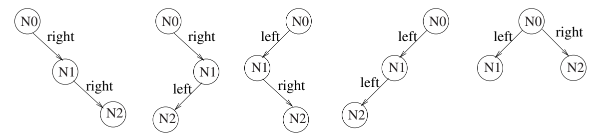

Figure 3: Trees generated for `finBinaryTree(3)`

Each object of the class `BinaryTree` represents a tree. The `size` field contains the number of nodes in the tree. Objects of the inner class `Node` represent nodes of the trees. The method `repOk` first checks if the tree is empty. If not, `repOk` traverses all nodes reachable from `root`, keeping track of the visited nodes to detect cycles. (The method `add` from `java.util.Set` returns `false` if the argument already exists in the set.)

To generate trees that have a given number of nodes, the Korat search algorithm uses the *finitization* description shown in Figure 2. The statements in the finitization description specify bounds on the number of objects to be used to construct instances of the data structure, as well as possible values stored in the fields of those objects. Most of the finitization description shown in the figure is automatically generated from the type declarations in the Java code. In Figure 2, the parameter `NUM_Node` specifies the bound on number of nodes in the tree. Each reference field in the tree is either `null` or points to one of the `Node` objects. Note that the identity of these objects is irrelevant—two trees are *isomorphic* if they have the same branching structure, irrespective of the actual nodes in the trees.

Korat automatically generates all nonisomorphic trees with a given number of nodes. For example, for `finBinaryTree(3)`, Korat generates the five trees shown in Figure 3. As another example, for `finBinaryTree(7)`, Korat generates 429 trees in less than one second.

We next illustrate how programmers can use Korat to check correctness of methods. The JML annotations in Figure 4 specify partial correctness for the example `remove` method that removes from a `BinaryTree` a node that is in the tree. The `normal_behavior` annotation specifies that if the precondition (`requires`) is satisfied at the beginning of the method, then the postcondition (`ensures`) is satisfied at the end of the method and the method returns without throwing an exception. (The helper method `has` checks that the tree contains the given node.) Implicitly, the class `invariant` is added to the precondition and the postcondition. Korat uses the JML tool-set to translate annotations into runtime Java assertions.

```
//@ public invariant repOk(); // class invariant
                              //   for BinaryTree
/*@ public normal_behavior    // specification for remove
  @     requires has(n);      //     precondition
  @     ensures !has(n);      //     postcondition
  @*/
public void remove(Node n) {
    // ... method body
}
```

Figure 4: Partial specification for `BinaryTree.remove`

```
public class HeapArray {
    private int size; // number of elements in the heap
    private Comparable[] array; // heap elements
    //@ public invariant repOk();
    public boolean repOk() {
        // checks that array is non-null
        if (array == null) return false;
        // checks that size is within array bounds
        if (size < 0 || size > array.length)
            return false;
        for (int i = 0; i < size; i++) {
            // checks that elements are non-null
            if (array[i] == null) return false;
            // checks that array is heapified
            if (i > 0 &&
                array[i].compareTo(array[(i-1)/2]) > 0)
                return false;
        }
        // checks that non-heap elements are null
        for (int i = size; i < array.length; i++)
            if (array[i] != null) return false;
        return true;
    }
}
```

Figure 5: `HeapArray` example

To test a method, Korat first generates test inputs. For `remove`, each input is a pair of a tree and a node. The precondition defines valid inputs for the method: the tree must be valid and the node must be in the tree. Given a finitization for inputs (which can be written reusing the finitization description for trees presented in Figure 2), Korat generates all nonisomorphic inputs. For `remove`, the number of input pairs is the product of the number of trees and the number of nodes in the trees. After generating the inputs, Korat invokes the method (with runtime assertions for postconditions) on each input and reports a counterexample if the method fails to satisfy the correctness criteria.

2.2 Heap array

This section illustrates the generation and checking of array-based data structures, using the heap data structure [8]. The (binary) *heap* data structure can be viewed as a complete binary tree—the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to some point. Heaps also satisfy the *heap property*—for every node n other than the root, the value of n's parent is greater than or equal to the value of n. The Java code in Figure 5 declares an array-based heap and defines the corresponding `repOk` method that checks if the input is a valid `HeapArray`.

The elements of the heap are stored in `array`. The elements implement the interface `Comparable`, providing the method `compareTo` for comparisons. The method `repOk` first checks for the special case when `array` is `null`. If not, `repOk` checks that the `size` of the heap is within the bounds of the `array`. Then, `repOk` checks that the array elements that belong to the heap are not `null` and that they satisfy the heap property. Finally, `repOk` checks that the array elements that do not belong to the heap are `null`.

```
public static Finitization finHeapArray(int MAX_size,
                                        int MAX_length,
                                        int MAX_elem) {
    Finitization f = new Finitization(HeapArray.class);
    // size in [0..MAX_size]
    f.set("size", new IntSet(0, MAX_size));
    f.set("array",
          // array.length in [0..MAX_length]
          new IntSet(0, MAX_length),
          // array[] in null + Integer([0..MAX_elem])
          new IntegerSet(0, MAX_elem).add(null));
    return f;
}
```

Figure 6: Finitization description for the `HeapArray` example

```
size = 0, array = []
size = 0, array = [null]
size = 1, array = [Integer(0)]
size = 1, array = [Integer(1)]
```

Figure 7: Heaps generated for `finHeapArray(1,1,1)`

To generate heaps, the Korat search algorithm uses the finitization description shown in Figure 6. Again, most of the finitization description shown in the figure is automatically generated from the type declarations in the Java code. In Figure 6, the parameters `MAX_size`, `MAX_length`, and `MAX_elem` bound the size of the heap, the length of the array, and the elements of the array, respectively. The elements of the array can either be `null` or contain `Integer` objects where the integers can range from 0 to `MAX_elem`.

Given values for the finitization parameters, Korat automatically generates all heaps. For example, for `finHeapArray(1,1,1)`, Korat generates the four heaps shown in Figure 7. As another example, in less than one second, for `finHeapArray(5,5,5)`, Korat generates 1919 heaps. Note that Korat requires only the `repOk` method (which can use the full Java language) and finitization to generate all heaps. Writing a dedicated generator for complex data structures [2] is much more involved than writing `repOk`.

We next illustrate how programmers can use Korat to check partial correctness of the `extractMax` method that removes and returns the largest element from a `HeapArray`. The JML annotations in Figure 8 specify partial correctness for the `extractMax` method. The `normal_behavior` specifies that if the input heap is valid and non-empty, then the method returns the largest element in the original heap and the resulting heap after execution of the method is valid. The JML keywords `\result` and `\old` denote, respectively, the object returned by the method and the expressions that should be evaluated in the pre-state. JML annotations can also express exceptional behavior of methods. The example `exceptional_behavior` specifies that if the input heap is empty, the method throws an `IllegalArgumentException`.

To check the method `extractMax`, Korat first uses a finitization to generate all nonisomorphic heaps that satisfy either the `normal_behavior` precondition or the `exceptional_behavior` precondition. Next, Korat invokes the method (with runtime assertions for postconditions) on each input and reports a counterexample if any invocation fails to satisfy the correctness criteria.

3. TEST CASE GENERATION

The heart of Korat is a technique for test case generation: given a Java predicate and a finitization for its input, Korat automatically generates all nonisomorphic inputs for which the predicate

```
/*@ public normal_behavior
  @     requires size > 0;
  @     ensures \result == \old(array[0]);
  @ also public exceptional_behavior
  @     requires size == 0;
  @     signals (IllegalArgumentException e) true;
  @*/
public Comparable extractMax() {
    // ... method body
}
```

Figure 8: Partial specification for `HeapArray.extractMax`

```
void koratSearch(Predicate p, Finitization f) {
    intialize(f);
    while (hasNextCandidate()) {
        Object candidate = nextCandidate();
        try {
            if (p.invoke(candidate))
                output(candidate);
        } catch (Throwable t) {}
        backtrack();
    }
}
```

Figure 9: Pseudo-code of the Korat search algorithm

returns `true`. Figure 9 gives an overview of the Korat search algorithm. The algorithm uses a *finitization* (described in Section 3.1) to bound the *state space* (Section 3.2) of predicate inputs. Korat uses backtracking (Section 3.3) to exhaustively 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 all the fields of the candidate input. To monitor the accesses, Korat instruments the predicate and all the methods that the predicate transitively invokes (Section 3.5). 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 cases (Section 3.4).

This section first illustrates how Korat generates valid inputs for predicate methods that take only the implicit `this` argument. Section 3.6 shows how Korat generates valid inputs for Java predicates that take multiple arguments.

3.1 Finitization

To generate a finite state space of a predicate's inputs, the search algorithm needs a *finitization*, i.e., 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 for each field the set of classes whose objects the field can point to. The set of values a field can take forms its *field domain*. Note that a field domain is a union of some class domains.

In the spirit of using the implementation language (which programmers are familiar with) for specification and testing, Korat provides a `Finitization` class that allows finitizations to be written in Java.[2] Korat automatically generates a finitization *skeleton* from the type declarations in the Java code. For the `BinaryTree` example presented in Figure 1, Korat automatically generates the skeleton shown in Figure 10.

[2]The initial version of Korat provided a special-purpose language for more compact descriptions of finitizations, sketched in the comments in the examples in Figures 2 and 6.

```
public static Finitization finBinaryTree(int NUM_Node,
                                          int MIN_size,
                                          int MAX_size) {
  Finitization f = new Finitization(BinaryTree.class);
  ObjSet nodes = f.createObjects("Node", NUM_Node);
  nodes.add(null);
  f.set("root", nodes);
  f.set("size", new IntSet(MIN_size, MAX_size));
  f.set("Node.left", nodes);
  f.set("Node.right", nodes);
  return f;
}
```

Figure 10: Generated finitization description for `BinaryTree`

In Figure 10, the `createObjects` method specifies that the input contains at most `NUM_Node` objects from the `Node`. The `set` method specifies the field domain for each field. In the skeleton, the fields `root`, `left`, and `right` are specified to contain either `null` or a `Node` object. The `size` field is specified to range between `MIN_size` and `MAX_size` using the utility class `IntSet`. The Korat package provides several additional classes for easy construction of class domains and field domains.

Once Korat generates a finitization skeleton, programmers can further specialize or generalize it. For example, the skeleton shown in Figure 10 can be specialized by setting `MIN_size` to 0 and `MAX_size` to `NUM_Node`. We presented another specialized finitization in Figure 2. Note that programmers can use the full expressive power of the Java language for writing finitization descriptions.

3.2 State space

We continue with the `BinaryTree` example to illustrate how Korat constructs the state space for the input to `repOk` using the finitization presented in Figure 2. Consider the case when Korat is invoked for `finBinaryTree(3)`, i.e., `NUM_Node = 3`. Korat first allocates the specified objects: one `BinaryTree` 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 eight elements; there are total of eight fields: the single `BinaryTree` object has two fields, `root` and `size`, and the three `Node` objects have two fields each, `left` and `right`.

For this example, a *candidate* `BinaryTree` input is a sample valuation of those eight fields. The state space of inputs consists of all possible assignments to those fields, where each field gets a value from its corresponding field domain. Since the domain for fields `root`, `left`, and `right` has four elements (`null` and three `Node`s from the `Node` class domain), the state space has $4 * 1 * (4 * 4)^3 = 2^{14}$ potential candidates. For `NUM_Node`$= n$, the state space has $(n + 1)^{2n+1}$ potential candidates. Figure 11 shows an example candidate that is a valid binary tree on three nodes. Not all valuations are valid binary trees. Figure 12 shows an example candidate that is not a tree; `repOk` returns `false` for this input.

3.3 Search

To systematically explore the state space, Korat orders all the elements in every class domain and every field domain (which is a union of class domains). 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.

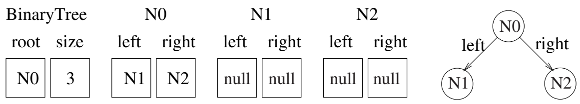

Figure 11: Candidate input that is a valid `BinaryTree`.

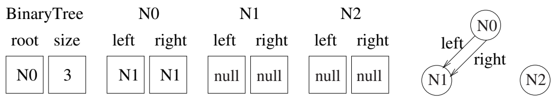

Figure 12: Candidate input that is not a valid `BinaryTree`.

Each candidate input is a vector of *field domain indices* into the corresponding field domains. For our running example with `NUM_Node = 3`, assume that the `Node` class domain is ordered as [N_0,N_1,N_2], and the field domains for `root`, `left`, and `right` are ordered as [`null`,N_0,N_1,N_2]. (`null` by itself forms a class domains.) The domain of the `size` field has a single element, 3. According to this ordering, the candidate inputs in Figures 11 and 12 have candidate vectors $[1, 0, 2, 3, 0, 0, 0, 0]$ and $[1, 0, 2, 2, 0, 0, 0, 0]$, respectively.

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 `repOk` to check the validity of the current candidate. During the execution of `repOk`, Korat monitors the fields that `repOk` accesses. Specifically, Korat builds a *field-ordering*: a list of the field identifiers ordered by the first time `repOk` accesses the corresponding field. Consider the invocation of `repOk` from Figure 1 on the candidate shown in Figure 12. In this case, `repOk` 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 `repOk` returns, Korat generates the next candidate vector backtracking on the fields accessed by `repOk`. 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. (The next section presents how Korat generates only nonisomorphic candidates by resetting a domain index for a field to zero even when the index does not exceed the size of the field domain.)

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 4^4 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 `repOk` did not read the other fields, and it could have returned `false` irrespective of the values of those fields.

Continuing further with our example, the next candidate is the valid tree shown in Figure 11. Before executing `repOk` on this candidate, Korat also initializes the field-ordering to `[0,2,3]`. Note that, if `repOk` accesses fields in a deterministic order, this is consistent with the first three fields that `repOk` is going to access, because the values of the first two fields in the field-ordering were not changed when constructing this candidate from the previous candidate. When `repOk` executes on this candidate, `repOk` returns `true` and the field-ordering that Korat builds is `[0,2,3,4,5,6,7,1]`. If `repOk` returns `true`, Korat outputs all (nonisomorphic) candidates that have the same values for the accessed fields as the current candidate. (Note that `repOk` may not access all reachable fields before returning `true`.) The search then backtracks to the next candidate.

Recall that Korat orders the values in the class and field domains. Additionally, each execution of `repOk` on a candidate imposes an order on the fields in the field-ordering. Together, these orders induce a lexicographic order on the candidates. The search algorithm described here generates inputs in the lexicographical order. Moreover, for non-deterministic `repOk` methods, our algorithm provides the following guarantee: all candidates for which `repOk` always returns `true` are generated; candidates for which `repOk` always returns `false` are never generated; and candidates for which `repOk` sometimes returns `true` and sometimes `false` may or may not be generated.

In practice, our search algorithm prunes large portions of the search space, and thus enables Korat to explore very large state spaces. The efficiency of the pruning depends on the `repOk` method. An ill-written `repOk`, for example, might always read the entire input before returning, thereby forcing Korat to explore almost every candidate. However, our experience indicates that naturally written `repOk` methods, which return `false` as soon as the first invariant violation is detected, induce very effective pruning.

3.4 Nonisomorphism

To further optimize the search, Korat avoids generating multiple candidates that are isomorphic to one another. Our optimization is based on the following definition of isomorphism.

Definition: 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. (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`) that the fields of objects in O can contain. Further, let $r \in O$ be a special root object, and let O_C be the set of all objects reachable from r in C. Two candidates, C and C', are *isomorphic* iff there is a permutation π on O, mapping objects from O_i to objects from O_i for all $1 \leq i \leq n$, such that:

$$\forall o, o' \in O_C.\ \forall f \in fields(o).\ \forall p \in P.$$
$$o.f{=}{=}o' \text{ in } C \text{ iff } \pi(o).f{=}{=}\pi(o') \text{ in } C' \text{ and}$$
$$o.f{=}{=}p \text{ in } C \text{ iff } \pi(o).f{=}{=}p \text{ in } C'.$$

The operator `==` is Java's comparison by object identity. Note that isomorphism is defined with respect to a root object. Two candidates are defined to be isomorphic if the parts of their object graphs reachable from the root object are isomorphic. In case of `repOk`, the root object is the `this` object that is passed as an implicit argument to `repOk`.

Isomorphism between candidates partitions the state space into *isomorphism partitions*. Recall the lexicographic ordering induced by the ordering on the values in the field domains and the field-orderings built by `repOk` executions. For each isomorphism partition, Korat generates only the lexicographically smallest candidate in that partition.

Conceptually, Korat avoids generating multiple candidates from the same isomorphism partition by incrementing field domain indices

```
class SomeClass {
    boolean somePredicate(X x, Y y) {...}
    ...
}
```

Figure 13: Predicate method with multiple arguments

by more than one: while backtracking on a field f in the field-ordering, Korat checks for how much to increment the field domain index of f as follows. Suppose that f contains a pointer to an object o_f that belongs to a class domain c_f. Recall that all objects in a class domain are ordered. Let i_f be the index of o_f in c_f. For instance, in the example ordering used above for `finBinaryTree(3)`, field domain index `2` for `right` corresponds to the class domain `Node` and class domain index `1`.

Further, Korat finds all fields f' such that f' occurs before f in the field-ordering and f' contains a pointer to an object o'_f of the same class domain c_f. Let i'_f be the index of o'_f in c_f, and let m_f be the maximum of all such indices i'_f. (If there is no such field f' before f in the field-ordering, $m_f =$`-1`.) In the example candidate for Figure 12, backtracking on $f =$`N0.right` gives $m_f =$`1`.

Then, during backtracking on f, Korat checks if i_f is greater than m_f. If $i_f \leq m_f$, Korat increments the field domain index of f by one. If $i_f > m_f$, Korat increments the field domain index of f so that it contains a pointer to an object of the class domain after c_f. If no such domain exists, i.e., c_f is the last domain for the field f, Korat resets the field domain index of f to zero and continues backtracking on the previous field in the field-ordering. The actual Korat implementation uses caching to speed up the computation of m_f.

For example, Korat for `finBinaryTree(3)` generates only the five trees shown in Figure 3. Each tree is a representative from an isomorphism partition that has six distinct trees, one for each of 3! permutations of nodes.

3.5 Instrumentation

To monitor `repOk`'s executions, Korat instruments all classes whose objects appear in finitizations by doing a source to source translation. For each of the classes, Korat adds a special constructor. For each field of those classes, Korat adds an identifier field and special `get` and `set` methods. In the code for `repOk` and all the methods that `repOk` transitively invokes, Korat replaces each field access with an invocation of the corresponding `get` or `set` 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* pattern [11]. Korat uses the special constructors to initialize all objects in a finitization with an observer. The search algorithm initializes each of the identifier fields to a unique index into the candidate vector. Special `get` and `set` 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).

3.6 Predicates with multiple arguments

The discussion so far described how Korat generates inputs that satisfy a `repOk` method. This section describes how Korat generalizes this technique to generate inputs that satisfy any Java predicate, including predicates that take multiple arguments. Figure 13 shows

```
class SomeClass_somePredicate {
    SomeClass This;
    X x;
    Y y;
    boolean repOk() {
        return This.somePredicate(x, y);
    }
}
```

Figure 14: Equivalent `repOk` method

a Java predicate that takes two arguments (besides `this`). In order to generate inputs for this predicate, Korat generates an equivalent `repOk` method shown in Figure 14. Korat then generates inputs to the `repOk` method using the technique described earlier.

4. TESTING METHODS

The previous section focused on automatic test case generation from a Java predicate and a finitization description. This section presents how Korat builds on this technique to check correctness of methods. Korat uses specification-based testing: to test a method, Korat first generates test inputs from the method's precondition, then invokes the method on each of those inputs, and finally checks the correctness of the output using the method's postcondition.

The current Korat implementation uses the Java Modeling Language (JML) [20] for specifications. Programmers can use JML annotations to express method preconditions and postconditions, as well as class invariants; these annotations use JML keywords `requires`, `ensures`, and `invariant`, respectively. Each annotation contains a boolean expression; JML uses Java syntax and semantics for expressions, and contains some extensions such as quantifiers. Korat uses a large subset of JML that can be automatically translated into Java predicates.

JML specifications can express several *normal* and *exceptional behaviors* for a method. Each behavior has a precondition and a postcondition: if the method is invoked with the precondition being satisfied, the behavior requires that the method terminate with the postcondition being satisfied. Additionally, normal behaviors require that the method return without an exception, whereas exceptional behaviors require that the method return with an exception. Korat generates inputs for all method behaviors using the *complete* method precondition that is a conjunction of: 1) the class invariant for all objects reachable from the input parameters and 2) a disjunction of the preconditions for all behaviors. In the text that follows, we refer to complete precondition simply as precondition.

4.1 Generating test cases

Valid test cases for a method must satisfy its precondition. To generate valid test cases, Korat uses a class that represents method's inputs. This class has one field for each parameter of the method (including the implicit `this` parameter) and a `repOk` predicate that uses the precondition to check the validity of method's inputs. Given a finitization, Korat then generates all inputs for which this `repOk` returns `true`; each of these inputs is a valid input to the original method.

We illustrate generation of test cases using the `remove` method for `BinaryTree` from Section 2. For this method, each input consists of a pair of `BinaryTree this` and a `Node n`, and the precondition is `this.has(n)`. Figure 15 shows the class that Korat uses for the method's inputs. For this class, Korat creates the finitization skeleton that reuses the finitization for `BinaryTree`, as shown in

```
class BinaryTree_remove {
    BinaryTree This;   // the implicit "this" parameter
    BinaryTree.Node n; // the Node parameter
    //@ invariant repOk();
    public boolean repOk() {
        return This.has(n);
    }
}
```

Figure 15: Class representing `BinaryTree.remove`

```
public static Finitization
        finBinaryTree_remove(int NUM_Node) {
    Finitization f =
        new Finitization(BinaryTree_remove.class);
    Finitization g = BinaryTree.finBinaryTree(NUM_Node);
    f.includeFinitization(g);
    f.set("This", g.getObjects(BinaryTree.class));
    f.set("n", /***/);
    return f;
}
```

Figure 16: Finitization skeleton for `BinaryTree_remove`

Figure 16. The comment `/***/` indicates that Korat cannot automatically determine an appropriate field domain for `n`.

To create finitization for `BinaryTree_remove`, the programmer modifies the skeleton, e.g., by replacing `/***/` with `g.get("root")` or `g.getObjects(BinaryTree.Node.class)` to set the domain for the parameter `n` to the domain for the field `root` or to the set of nodes from the finitization `g`, respectively. Given a value for `NUM_Node`, Korat then generates all valid test cases, each of which is a pair of a tree (with the given number of nodes) and a node from that tree.

4.1.1 Dependent and independent parameters

For the `remove` method, the precondition makes the parameters `This` and `n` explicitly dependent. When the parameters are independent, programmers can instruct Korat to generate all test cases by separately generating all possibilities for each parameter and creating all valid test cases as the Cartesian product of these possibilities.

We next compare Korat with another approach for generating all valid (nonisomorphic) test cases, which uses the Cartesian product even for dependent parameters. Consider a method `m`, with n parameters and precondition $\mathtt{m}_{pre}$. Suppose that a set of possibilities S_i, $1 \leq i \leq n$, is given for each of the parameters. All valid test cases from $S_1 \times \ldots \times S_n$ can be then generated by creating *all* n-tuples from the product, followed by filtering each of them through $\mathtt{m}_{pre}$. (This approach is used in the JML+JUnit testing framework [6] that combines JML [20] and JUnit [3].) Note that this approach requires manually constructing possibilities for all parameters, some of which can be complex data structures.

Korat, on the other hand, constructs data structures from a simple description of the fields in the structures. Further, in terms of Korat's search of `repOk`'s state space, the presented approach would correspond to the search that tries every candidate input. Korat improves on this approach by: 1) pruning the search based on the accessed fields and 2) generating only one representative from each isomorphism partition.

4.2 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. Programmers could also manually develop more elaborate test oracles. Programmers can also check for properties that relate the post-state with the *pre-state* (i.e., the state just before the method's invocation).

	Testing framework		
testing activity	JUnit	JML+JUnit	Korat
generating test cases			$\surd$
generating test oracle		$\surd$	$\surd$
running tests	$\surd$	$\surd$	$\surd$

Table 1: Comparison of several testing frameworks for Java. Automated testing activities are indicated with "$\surd$".

The current Korat implementation uses the JML tool-set to automatically generate test oracles from method postconditions, as in the JML+JUnit framework [6]. The JML tool-set translates JML postconditions into runtime Java assertions. If an execution of a method violates such an assertion, an exception is thrown 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 of the JML tool-set and translation can be found in [20].

Korat also uses JML+JUnit to combine JML test oracles with JUnit [3], 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. JML+JUnit, thus, automates both test execution and correctness checking. However, JML+JUnit requires programmers to provide sets of possibilities for all method parameters: it generates all valid inputs by generating the Cartesian product of possibilities and filtering the tuples using preconditions. Korat additionally automates generation of test cases, thus automating the entire testing process. Table 1 summarizes the comparison of these testing frameworks.

5. EXPERIMENTAL RESULTS

This section presents the performance results of the Korat prototype. We used Java to implement the search for valid nonisomorphic `repOk` inputs. For automatic instrumentation of `repOk` (and transitively invoked methods), we modified the sources of the Sun's `javac` compiler. We also modified `javac` to automatically generate finitization skeletons. For checking method correctness, we slightly modified the JML tool-set, building on the existing JML+JUnit framework [6].

We first present Korat's performance for test case generation, then compare Korat with the test generation that uses Alloy Analyzer [16], and finally present Korat's performance for checking method correctness. We performed all experiments on a Linux machine with a Pentium III 800 MHz processor using Sun's Java 2 SDK1.3.1 JVM.

benchmark	package	finitization parameters
`BinaryTree`	`korat.examples`	NUM_Node
`HeapArray`	`korat.examples`	MAX_size, MAX_length, MAX_elem
`LinkedList`	`java.util`	MIN_size, MAX_size, NUM_Entry, NUM_Object
`TreeMap`	`java.util`	MIN_size, NUM_Entry, MAX_key, MAX_value
`HashSet`	`java.util`	MAX_capacity, MAX_count, MAX_hash, loadFactor
`AVTree`	`ins.namespace`	NUM_AVPair, MAX_child, NUM_String

Table 2: Benchmarks and finitization parameters. Each benchmark is named after the class for which data structures are generated; the structures also contain objects from other classes.

5.1 Benchmarks

Table 2 lists the benchmarks for which we show Korat's performance. `BinaryTree` and `HeapArray` are presented in Section 2. (Additionally, `HeapArray`s are similar to array-based stacks and queues, as well as `java.util.Vector`s.) `LinkedList` is the implementation of linked lists in the Java Collections Framework, a part of the standard Java libraries. This implementation uses doubly-linked, circular lists that have a `size` field and a `header` node as a sentinel node. (Linked lists also provide methods that allow them to be used as stacks and queues.) `TreeMap` implements the `Map` interface using red-black trees [8]. This implementation uses binary trees with `parent` fields. Each node (implemented with inner class `Entry`) also has a `key` and a `value`. (Setting all `value` fields to `null` corresponds to the set implementation in `java.util.TreeSet`.) `HashSet` implements the `Set` interface, backed by a hash table [8]. This implementation builds collision lists for buckets with the same hash code. The `loadFactor` parameter determines when to increase the size of the hash table and rehash the elements.

`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. INS uses attributes and values to classify services based on their properties. The names of these properties are implemented with arbitrary `String`s except that `"*"` is a wildcard that matches all other values. The finitization bounds the number of `AVPair` objects that implement nodes, the number of children for each node, and the total number of `String`s (including the wildcard).

5.2 Korat's test case generation

Table 3 presents the results for generating valid structures with our Korat implementation. For each benchmark, all finitization parameters are set to the same (size) value (except the `loadFactor` parameter for `HashSet`, which is set to default 0.75). For a range of size values, we tabulate the time that Korat takes to generate all valid structures, the number of structures generated, the number of candidate structures checked by `repOk`, and the size of the state space.

Korat can generate all structures even for very large state spaces because the search pruning allows Korat to explore only a tiny fraction of the state space. The ratios of the number of candidate

benchmark	size	time (sec)	structures generated	candidates considered	state space
`BinaryTree`	8	1.53	1430	54418	2^{53}
	9	3.97	4862	210444	2^{63}
	10	14.41	16796	815100	2^{72}
	11	56.21	58786	3162018	2^{82}
	12	233.59	208012	12284830	2^{92}
`HeapArray`	6	1.21	13139	64533	2^{20}
	7	5.21	117562	519968	2^{25}
	8	42.61	1005075	5231385	2^{29}
`LinkedList`	8	1.32	4140	5455	2^{91}
	9	3.58	21147	26635	2^{105}
	10	16.73	115975	142646	2^{120}
	11	101.75	678570	821255	2^{135}
	12	690.00	4213597	5034894	2^{150}
`TreeMap`	7	8.81	35	256763	2^{92}
	8	90.93	64	2479398	2^{111}
	9	2148.50	122	50209400	2^{130}
`HashSet`	7	3.71	2386	193200	2^{119}
	8	16.68	9355	908568	2^{142}
	9	56.71	26687	3004597	2^{166}
	10	208.86	79451	10029045	2^{190}
	11	926.71	277387	39075006	2^{215}
`AVTree`	5	62.05	598358	1330628	2^{50}

Table 3: Korat's performance on several benchmarks. All finitization parameters are set to the size value. Time is the elapsed real time in seconds for the entire generation. State size is rounded to the nearest smaller exponent of two.

structures considered and the size of the state spaces show that the key to effective pruning is backtracking based on fields accessed during `repOk`'s executions. Without backtracking, and even with isomorphism optimization, Korat would generate infeasibly many candidates. Isomorphism optimization further reduces the number of candidates, but it mainly reduces the number of valid structures.

For `BinaryTree`, `LinkedList`, `TreeMap`, and `HashSet` (with the `loadFactor` parameter of 1), the numbers of nonisomorphic structures appear in the Sloane's On-Line Encyclopedia of Integer Sequences [30]. For all these benchmarks, Korat generates exactly the actual number of structures.

5.2.1 Comparison with Alloy Analyzer

We next compare Korat's test case generation with that of the Alloy Analyzer (AA) [16], an automatic tool for analyzing Alloy *models*. Alloy [17] is a first-order, declarative language based on relations. Alloy is suitable for modeling structural properties of software. Alloy models of several data structures can be found in [22]. These models specify class invariants in Alloy, which correspond to `repOk` methods in Korat, and also declare field types, which corresponds to setting field domains in Korat finitizations.

Given a model of a data structure and a *scope*—a bound on the number of atoms in the universe of discourse—AA can generate all (mostly nonisomorphic) *instances* of the model. An instance valuates the relations in the model such that all constraints of the model are satisfied. Setting the scope in Alloy corresponds to setting the finitization parameters in Korat. AA translates the input Alloy model into a boolean formula and uses an off-the-shelf SAT solver to find a satisfying assignment to the formula. Each such assignment is translated back to an instance of the input model. AA adds symmetry-breaking predicates [29] to the boolean formula so that different satisfying assignments to the formula represent (mostly) nonisomorphic instances of the input model.

		Korat			Alloy Analyzer		
benchmark	size	struc. gen.	total time	first struc.	inst. gen.	total time	first inst.
BinaryTree	3	5	0.56	0.62	6	2.63	2.63
	4	14	0.58	0.62	28	3.91	2.78
	5	42	0.69	0.67	127	24.42	4.21
	6	132	0.79	0.66	643	269.99	6.78
	7	429	0.97	0.62	3469	3322.13	12.86
HeapArray	3	66	0.53	0.58	78	11.99	6.20
	4	320	0.57	0.59	889	171.03	16.13
	5	1919	0.73	0.63	1919	473.51	39.58
LinkedList	3	5	0.58	0.60	10	2.61	2.39
	4	15	0.55	0.65	46	3.47	2.77
	5	52	0.57	0.65	324	14.09	3.51
	6	203	0.73	0.61	2777	148.73	5.74
	7	877	0.87	0.61	27719	2176.44	10.51
TreeMap	4	8	0.75	0.69	16	12.10	6.35
	5	14	0.87	0.88	42	98.09	18.08
	6	20	1.49	0.98	152	1351.50	50.87
AVTree	2	2	0.55	0.65	2	2.35	2.43
	3	84	0.65	0.61	132	4.25	2.76
	4	5923	1.41	0.61	20701	504.12	3.06

Table 4: Performance comparison. For each benchmark, performances of Korat and AA are compared for a range of finitization values. For values larger than presented, AA does not complete its generation within 1 hour. Korat's performance for larger values is given in Table 3.

Table 4 summarizes the performance comparison. Since AA cannot handle arbitrary arithmetic, we do not generate `HashSets` with AA. For all other benchmarks, we compare the total number of structures/instances and the time to generate them for a range of parameter values. We also compare the time to generate the first structure/instance.

Time presented is the total elapsed real time (in seconds) that each experiment took from the beginning to the end, including start-up.[3] Start-up time for Korat is approximately 0.5 sec. (That is why in some cases it seems that generating all structures is faster than generating the first structure or that generating all structures for a larger input is faster than generating all structures for a smaller input.) Start-up time for AA is somewhat higher, approximately 2 sec, as AA needs to translate the model and to start a SAT solver. AA uses precompiled binaries for SAT solvers.

In all cases, Korat outperforms AA; Korat is not only faster for smaller inputs, but it also completes generation for larger inputs than AA. There are two reasons that could account for this difference. Since AA translates Alloy models into boolean formulas, it could be that the current (implementation of the) translation generates unnecessarily large boolean formulas. Another reason is that often AA generates a much greater number of instances than Korat, which takes a greater amount of time by itself. One way to reduce the number of instances generated by AA is to add more symmetry-breaking predicates.

Our main argument for developing Korat was simple: for Java programmers not familiar with Alloy, it is easier to write a `repOk` method than an Alloy model. (From our experience, for researchers familiar with Alloy, it is sometimes easier to write an Alloy model than a `repOk` method.) Before conducting the above experiments, we expected that Korat would generate structures slower than AA.

[3]We include start-up time, because AA does not provide generation time only for generating all instances. We eliminate the effect of cold start by executing each test twice and taking the smaller time.

benchmark	method	max. size	test cases generated	gen. time	test time
BinaryTree	remove	3	15	0.64	0.73
HeapArray	extractMax	6	13139	0.87	1.39
LinkedList	reverse	2	8	0.67	0.76
TreeMap	put	8	19912	136.19	2.70
HashSet	add	7	13106	3.90	1.72
AVTree	lookup	4	27734	4.33	14.63

Table 5: Korat's performance on several methods. All upper-limiting finitization parameters for method inputs are set to the given maximum size. These sizes give complete statement coverage. Times are the elapsed real times in seconds for the entire generation of all valid test cases and testing of methods for all those inputs. These times include writing and reading of files with test cases.

Our intuition was that Korat depends on the executions of `repOk` to "learn" the invariants of the structures, whereas AA uses a SAT solver that can "inspect" the entire formula (representing invariants) to decide how to search for an assignment. The experimental results show that our assumption was incorrect—Korat generates structures much faster than AA. We are now exploring a translation of Alloy models into Java (or even C) and the use of Korat (or a similar search) to generate instances.

5.3 Checking correctness

Table 5 presents the results for checking methods with Korat. For each benchmark, a representative method is chosen; the results are similar for other methods. Methods `remove` and `extractMax` are presented in Section 2. Method `reverse`, from `java.util.Collections`, uses list iterators to reverse the order of list elements; this method is static. Method `put`, from `java.util.TreeMap`, inserts a key-value pair into the map; this method has three parameters (`this`, `key`, and `value`) and invokes several helper methods that rebalance the tree after insertion. Method `add` inserts an element into the set. Method `lookup`, from INS, searches a `database` of intentional names for a given `query` intentional name. The correctness specifications for all methods specify simple containment properties (beside preservation of class invariants).

For each method, the `MIN` finitization parameters are set to zero and the `MAX` and `NUM` parameters to the same size value. Thus, the methods are checked for all valid inputs up to the maximum size, not only for the maximum size. The results show that it is practical to use Korat to exhaustively check correctness of intricate methods that manipulate complex data structures.

AA can also be used to check correctness of Java methods by writing method specifications as Alloy models and defining appropriate translations between Alloy instances and Java objects, as demonstrated in the TestEra framework [22]. However, the large number of instances generated by AA makes TestEra less practical to use than Korat. For example, maximum sizes six and eight for `extractMax` and `put` methods, respectively, are the smallest that give complete statement coverage. As shown in Table 4, for these sizes, AA cannot in a reasonable time even generate data structures that are parts of the inputs for these methods.

6. RELATED WORK

6.1 Specification-based testing

There is a large body of research on specification-based testing. An early paper by Goodenough and Gerhart [13] emphasizes its importance. Many projects automate test case generation from specifications, such as Z specifications [15, 31], UML statecharts [25, 26], or ADL specifications [5, 28]. These specifications typically do not consider linked data structures, and the tools do not generate Java test cases.

The TestEra framework [22] generates Java test cases from Alloy [17] specifications of linked data structures. TestEra uses the Alloy Analyzer (AA) [16] to automatically generate method inputs and check correctness of outputs, but it requires programmers to learn a specification language much different than Java. Korat generates inputs directly from Java predicates and uses the Java Modeling Language (JML) [20] for specifications. The experimental results also show that Korat generates test cases faster and for larger scopes than AA.

Cheon and Leavens [6] describe automatic translation of JML specifications into test oracles for JUnit [3]. This framework automates execution and checking of methods. However, the burden of test case generation is still on programmers: they have to provide sets of possibilities for all method parameters. Korat builds on this framework by automating test case generation.

6.2 Static analysis

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [10] uses a theorem prover to verify partial correctness of classes annotated with JML specifications. ESC has been used to verify absence of such errors as null pointer dereferences, array bounds violations, and division by zero. However, tools like ESC cannot verify properties of complex linked data structures.

There are some recent research projects that attempt to address this issue. The Three-Valued-Logic Analyzer (TVLA) [27] is the first static analysis system to verify that the list structure is preserved in programs that perform list reversals via destructive updating of the input list. 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) [24] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers that do not add extra information. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. While TVLA and PALE are primarily intraprocedural, Role Analysis [19] 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.

Jackson and Vaziri propose an approach [18] for analyzing methods that manipulate linked data structures. Their approach is to first build an Alloy model of bounded initial segments of computation sequences and then check the model exhaustively with AA. This approach provides static analysis, but it is unsound with respect to both the size of input and the length of computation. Korat not only checks the entire computation, but also handles larger inputs and more complex data structures than those in [18]. Further, Korat does not require Alloy, but JML specifications, and more importantly, unlike [18], Korat does not require specifications for all (helper) methods.

6.3 Software model checking

There has been a lot of recent interest in applying model checking to software. JavaPathFinder [32] and VeriSoft [12] operate directly on a Java, respectively C, program and systematically explore its state to check correctness. Other projects, such as Bandera [7] and JCAT [9], translate Java programs into the input language of existing model checkers like SPIN [14] and SMV [23]. 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.

7. CONCLUSIONS

This paper presented Korat, a novel framework for automated testing of Java programs. Given a formal specification for a method, Korat uses the method precondition to automatically generate all nonisomorphic test cases up to a given small size. Korat then executes the method on each test case, and uses the method postcondition as a test oracle to check the correctness of each output.

To generate test cases for a method, Korat constructs a Java predicate (i.e., a method that returns a boolean) from the method's precondition. The heart of Korat is a technique for automatic test case generation: given a predicate and a finitization for its inputs, Korat generates all nonisomorphic inputs for which the predicate returns `true`. Korat exhaustively explores the input space of the predicate, but does so efficiently by: 1) monitoring the predicate's executions to prune large portions of the search space and 2) generating only nonisomorphic inputs.

The Korat prototype uses the Java Modeling Language (JML) for specifications, i.e., class invariants and method preconditions and postconditions. Good programming practice suggests that implementations of abstract data types should already provide methods for checking class invariants—Korat then generates test cases almost for free.

This paper illustrated the use of Korat for testing several data structures, including some from the Java Collections Framework. The experimental results show that it is feasible to generate test cases from Java predicates, even when the search space for inputs is very large. This paper also compared Korat with the Alloy Analyzer, which can be used to generate test cases from declarative predicates. Contrary to our initial expectation, the experiments show that Korat generates test cases much faster than the Alloy Analyzer.

Acknowledgements

We would like to thank Michael Ernst, Daniel Jackson, Alexandru Sălcianu, and the anonymous referees for their comments on this paper. We are also grateful to Viktor Kuncak for helpful discussions on Korat and Alexandr Andoni for helping us with experiments. This work was funded in part by NSF grant CCR00-86154.

8. 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 (SOSP)*, Kiawah Island, Dec. 1999.

[2] 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.

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

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

[5] 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.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01-12, Department of Computer Science, Iowa State University, Nov. 2001.

[7] 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.

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

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

[10] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[12] 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.

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

[14] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[22] 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.

[23] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[24] 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.

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

[26] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

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

[28] 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.

[29] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[30] 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`.

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

[32] 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.