Optimizing Generation of Object Graphs in Java PathFinder

Milos Gligoric Tihomir Gvero
University of Belgrade
Belgrade, Serbia
{milos.gligoric,tihomir.gvero}@gmail.com

Steven Lauterburg Darko Marinov
University of Illinois
Urbana, IL 61801
{slauter2,marinov}@cs.uiuc.edu

Sarfraz Khurshid
University of Texas
Austin, TX 78712
khurshid@ece.utexas.edu

Abstract

Java PathFinder (JPF) is a popular model checker for Java programs. JPF was used to generate object graphs as test inputs for object-oriented programs. Specifically, JPF was used as an implementation engine for the Korat algorithm. Korat takes two inputs—a Java predicate that encodes properties of desired object graphs and a bound on the size of the graph—and generates all graphs (within the given bound) that satisfy the encoded properties. Korat uses a systematic search to explore the bounded state space of object graphs. Korat search was originally implemented in JPF using a simple instrumentation of the Java predicate. However, JPF is a general-purpose model checker and such direct implementation results in an unnecessarily slow search. We present our results on speeding up Korat search in JPF. The experiments on ten data structure subjects show that our modifications of JPF reduce the search time by over an order of magnitude.

1. Introduction

Software testing is important but time-consuming and expensive. Automated testing can significantly improve testing and reduce its cost. We developed an approach [6, 24–26] for automated, constraint-based test generation: the user manually encodes only the constraints for the desired properties of test inputs, and a tool automatically generates a large number of inputs. Our approach was used to test several applications, e.g., a network protocol and a constraint analyzer developed in academia [25], a fault-tree analyzer developed for NASA [30], an XPath compiler at Microsoft [29], and a web traversal application at Google [26]. Generation tools for our approach were mostly *developed from scratch*. However, they are based on backtracking search and could thus *leverage existing tools*, such as model checkers, that support backtracking.

Java PathFinder (JPF) is a popular, explicit-state model checker for Java programs [1, 32]. JPF implements a customized Java Virtual Machine (JVM) that supports state backtracking and control over non-deterministic choices (including thread scheduling). The customized JVM allows JPF to directly check properties of a wide range of Java programs without requiring the users to build models in a specialized language. Properties of control have been the traditional focus of model checking [7], with data often abstracted away. Recent progress on model checking has introduced several new techniques that can check complex properties of data (in addition to properties of control) [4, 5, 11, 20, 33–35]. Specifically, JPF was used to generate *object graphs*—where nodes are objects and edges are pointers between objects—which can represent test inputs for object-oriented programs [20, 33].

In this paper, we explore the use of JPF as an implementation engine for the Korat algorithm [6, 24]. Korat takes two inputs—a Java predicate that encodes constraints of desired object graphs and a bound on the size of the graphs—and generates all graphs (within the given bound) for which the predicate returns true. Korat uses a systematic search to explore the bounded state space of object graphs. We initially implemented Korat in JPF using a simple instrumentation of Java code, following the lazy initialization proposed by Khurshid et al. [20]. However, JPF is a general-purpose model checker and such direct implementation makes generation of object graphs unnecessarily slow.

The key question we asked is how a general tool such as JPF needs to be modified to *efficiently* implement an algorithm such as Korat search. While we focus on JPF and Korat, we believe that our work highlights important practical considerations when using model checkers to implement search engines in general, not only for test generation but also in other domains, e.g., in data structure repair [13, 14]. The goal of our work is not to obtain the most efficient implementation of Korat in general; in fact, the results in Section 5.2.2 show that even the most optimized version of Korat on JPF still runs slower than our publicly available version of Korat running on a regular JVM [2].

We present our work on speeding up Korat in JPF. JPF performs three core operations: (1) executing bytecodes,

(2) storing and restoring program states during backtracking, and (3) comparing states. Our modifications target each of these operations and enable efficient generation of object graphs using JPF. We take a pragmatic approach to modifying JPF: we profile JPF for a set of subject programs and then devise ways to address the largest bottleneck. The modifications and customizations we made in JPF range from trivial setting of existing flags to more complex implementing of optimized libraries to fairly extensive adding of new features to speed up execution and backtracking of operations that manipulate heaps, stacks, and threads. Our optimizations include controlled garbage collection, undo operations for heap backtracking, dynamic analyses for faster stack manipulations, simplified versions of thread operations, lightweight collections, and selective delegation of bytecode execution to the underlying JVM. Two of our modifications, while motivated by Korat, apply more broadly and were included in the publicly available JPF codebase [1, 17].

Various other projects also investigated how to speed up backtracking. Most recently, Xu et al. [36] present an efficient technique for checkpointing and replaying Java program execution. They focus on debugging of long-running programs, and their technique is very effective for restoring states when execution between choice points is long and choice points are infrequent. However, Korat search has very frequent choice points and short execution between them. We also developed a technique based on *abstract undo* which enables Korat to be executed faster on a regular JVM [14]. That technique uses special libraries where operations can be performed in reverse (e.g., to undo an insertion of an element into a set, one should remove the element from the set). The work in this paper presents a series of optimizations applicable to JPF not a regular JVM.

This paper makes the following contributions:

- **Solving imperative constraints using a model checker:** We use JPF to directly implement the Korat search by modifying the bytecode interpreter in JPF.

- **Speeding-up search:** We describe a suite of modifications to JPF that synergistically enable a significant speed-up in the search time. Two of these modifications apply more broadly than just for Korat and were included in JPF [1, 17].

- **Evaluation:** We evaluate our modifications to JPF using ten data structure subjects. The experimental results show significant speed-ups in the performance of JPF, ranging from 10.58x to 31.29x, with a median of 16.18x, i.e., over an order of magnitude in each case.

2. Example

We first illustrate *what* the search algorithm in Korat does, i.e., what its input and output are. As our running example, we use a set implemented as a red-black tree [8], an often-used example for test generation [3, 10, 23, 25, 27, 34, 35]. Figure 1 shows the relevant Java code. Each object of the class `RedBlackTree` represents a red-black tree, and each object of the class `Node` represents a node. The tree points to the `root` node and has a number of elements (`size`). Each node stores an `element`, has a certain `color` (red or black), and has pointers to its `left` and `right` children and its `parent`.

To generate object graphs that represent red-black trees, Korat requires that the user provides (1) a method that checks validity of object graphs and (2) a set of bounds for the size of the graphs. Figure 1 shows the method `repOk` that checks whether an object graph reachable from the `RedBlackTree` object (given as the implicit parameter `this`) indeed represents a valid red-black tree. We call these methods, which effectively check the representation invariant of a data structure, *`repOk` methods* [6, 22], and we call object graphs for which `repOk` returns `true` *valid*. Our example method invokes three helper methods to check that the object graph reachable from `root` is indeed a tree (has no sharing between objects), has nodes of proper colors (red nodes should have black children, and number of black nodes should be the same on all paths from root to the leaf [8]), and has elements ordered for binary search (elements in the left subtree should be smaller than the current element, and elements in the right subtree should be larger).

Figure 1 also presents a part of the `isTree` method. This method uses a breadth-first traversal to check that nodes reachable from `root` following `left` and `right` fields form a tree (and also checks that `parent` is inverse of the children and `size` has the appropriate value). The method uses a `workList` to traverse all nodes and keeps track of the `visited` nodes to detect sharing between nodes. (The method invocation `s.add(e)` returns `false` if the element `e` is already in the set `s`.) These methods are fairly easy to write, and junior researchers (such as the first two paper authors) with no prior experience with Korat can write them in a matter of few hours.

We refer to the other input to Korat [6], which provides bounds for object graphs, as *finitization*. Each finitization provides (1) a bound for the number of objects of each class in one graph and (2) a set of values for each field of those objects. For `RedBlackTree`, our example finitization specifies one `RedBlackTree` object (since Korat generates one tree at a time), some number N of `Node` objects, and these field values: `root`, `left`, `right`, and `parent` are either `null` or point to one of the N `Node` objects; `color` is either red or black; `element` ranges between 1 and N; and `size` is set to

```
public class RedBlackTree {
  Node root;
  int size;

  static class Node {
    int element;
    boolean color;
    Node left, right, parent;
    ...
  }

  boolean repOk() {
    if (!isTree()) return false;
    if (!hasProperColors()) return false;
    if (!areElementsOrdered()) return false;
    return true;
  }
  boolean isTree() {
    if (root == null) return size == 0; // is size correct
    if (root.parent != null) return false; // incorrect parent
    Set<Node> visited = new Set<Node>();
    visited.add(root);
    Queue<Node> workList = new Queue<Node>();
    workList.add(root);
    while (!workList.isEmpty()) {
      Node current = workList.remove();
      if (current.left != null) {
        if (!visited.add(current.left))
          return false; // not a tree
        if (current.left.parent != current)
          return false; // incorrect parent
        workList.add(current.left);
      }
      ... // analogous for current.right
    }
    return visited.size() == size; // is size correct
  }
  ...
}
```

Figure 1. Code that declares red-black trees and checks their validity

exactly N (with the goal to generate trees with exactly N nodes; another typical scenario is to generate trees whose size ranges between 0 and N). The users write finitizations using a small Java library for Korat; detailed examples are available elsewhere [6, 24, 26].

The goal of Korat it to generate *all valid* object graphs, within the bounds given in the finitization, i.e., all graphs for which the `repOk` method returns `true`. In our example, it means that Korat generates all red-black trees that have exactly N nodes and whose elements range between 1 and N. More precisely, the goal is to generate only *non-isomorphic* object graphs, where two object graphs are defined to be isomorphic if they differ only in the identity of the objects in the graph but have the same shape and the same values for primitive fields [6, 18, 24]. We want to avoid isomorphic object graphs since they constitute equivalent inputs for testing: they either both reveal a bug or none of them reveals a bug. Note that each red-black tree is translated into a test input, for example to test an operation that removes an element from the tree. While some properties of such operations can be checked statically, e.g., using shape analysis [28, 37], and would not require testing, other properties are beyond the reach of current static analyses and require testing as shown elsewhere [25, 26, 29, 30].

For our example, there are 122 non-isomorphic red-black trees of size 9. However, the state space for $N = 9$ is quite large: `root` has 10 possible values (either `null` or point to one of the 9 nodes), and each of the 9 nodes has 10 possible values for `left`, `right`, and `parent`, 9 possible values for `element`, and 2 possible values for `color`; thus, the state space contains $10 \cdot (10 \cdot 10 \cdot 10 \cdot 9 \cdot 2)^9 \sim 2 \cdot 10^{39}$ object graphs. We need an efficient search algorithm to identify 122 non-isomorphic red-black trees among these $2 \cdot 10^{39}$ object graphs [6].

3. Basic search in JPF

We next illustrate *how* the search algorithm in Korat works. We also describe how Korat was implemented in JPF using code instrumentation [20].

3.1. Korat search

Korat implements a backtracking search algorithm that systematically explores the space of `repOk`'s inputs, bounded by the finitization. Korat efficiently prunes the search through the space to generate only the non-isomorphic valid inputs, but all of them within the given bounds. We briefly describe how Korat generates such inputs. We present only the parts of Korat necessary to describe how to implement it in JPF; more details of Korat are available elsewhere [6, 24, 26].

The main entity in Korat's search is a *candidate* object graph, which may or may not be valid. Korat uses the finitization to construct a *domain* (i.e., an ordered set) of values for each object field of a candidate. For example, for generating red-black trees with nine nodes, the domain for `root` is `[null, N0, ..., N8]` where `N0, ..., N8` represent the nine node objects.

Korat starts its search from the initial candidate that has each of its object fields initialized to the first value in the corresponding domain. For example, `root` is initially `null`.

Korat generates candidates in a loop until it explores the entire space of the method's inputs. The basic idea is to execute the `repOK` method on a candidate to determine (1) whether the candidate is valid or not, and (2) what next candidate to try out. If `repOK` returns `true`, the object graph is valid and Korat outputs it. Regardless of whether `repOK` returns `true` or `false`, Korat continues its search with the next candidate.

During the execution of `repOK`, Korat monitors field accesses to record the order of their first access by `repOk`. Korat uses this order to prune the search. Indeed, the key idea in Korat is to generate the next candidate based on the

repOk's execution on the previous candidate. The intuition is that if repOK returns false by accessing only some fields of the object graph, then it would return false for any values of the non-accessed fields. Therefore, Korat backtracks the search based on the last accessed field. Korat tries the next value from the domain for that field to generate the next candidate. To eliminate exploration of isomorphic structures, Korat skips certain values from field domains when possible [6].

3.2. Korat search in JPF

JPF's backtracking enables a fairly straightforward implementation of Korat using code instrumentation [20]. The JPF library class Verify provides non-deterministic choice. The method Verify.getInt(int lo, int hi) returns non-deterministically a value between lo and hi (similar to VS_toss in VeriSoft [16]). The exploration of all candidates invokes getInt for each object field and uses the returned value as an index into the domain of possible values for the field. The monitoring of field accesses uses *shadow* boolean fields: introduce a new boolean field for each existing object field, set the new field initially to false, and set it to true when the object field is accessed for the first time.

Search is restricted to non-isomorphic structures by tracking *assigned* objects of a field domain, i.e., objects that are pointed to by some field in the structure constructed thus far. By restricting field assignments to (at most) one new object (for each type) that is not included in the structure thus far guarantees that exploration only considers non-isomorphic structures [6, 20, 24].

To illustrate, Figure 2 shows the code from Figure 1 instrumented for execution on JPF's JVM using JPF libraries. Note that to enable Korat search, the instrumentation introduces new fields and methods as well as replaces field accesses in repOk (and its helper methods) by method invocations of appropriate get methods. A key functionality that the get methods provide is monitoring of field accesses using the shadow fields. Additionally, the use of getInt enables the creation of next candidates, as well as an implicit maintenance of the order of field accesses.

Our basic implementation of Korat in JPF is related to but slightly differs from *lazy initialization* [20] of reference fields, where the first access of an object field non-deterministically initializes it to:

- null;
- an object (of appropriate type) that exists in the structure constructed thus far; or
- a new object (of appropriate type).

```
public class RedBlackTree {
  // data for finitization and search
  static Node[] nodes;
  static int assigned_nodes;
  static int min_size;
  static int max_size;
  // instrumentation for "root" field
  Node root;
  boolean init_root = false;
  Node get_root() {
    if (!init_root) {
      init_root = true;
      int i = Verify.getInt(0, min(assigned_nodes + 1,
                                   nodes.length - 1));
      if (i == assigned_nodes + 1)
        assigned_nodes++;
      root = nodes[i];
    }
    return root;
  }
  // instrumentation for "size" field
  int size;
  boolean init_size = false;
  int get_size() {
    if (!init_size) {
      init_size = true;
      size = Verify.getInt(min_size, max_size);
    }
    return size;
  }

  static class Node {
    ... // analogous instrumentation for each field
  }

  boolean repOk() {
    ... // same as in Fig. 1
  }
  boolean isTree() {
    if (get_root() == null) return get_size() == 0;
    if (get_root().get_parent() != null) return false;
    ... // replace read of each field "f"
        // with a call to "get_f" method
  }

  // "N" is the bound for finitization
  static void main(int N) {
    nodes = new Node[N + 1]; // nodes[0] = null;
    for (int i=1; i < nodes.length; i++) nodes[i] = new Node();
    min_size = max_size = N;
    RedBlackTree rbt = new RedBlackTree();
    // this call to "repOk" will backtrack a number of times,
    // setting the fields of objects reachable from "rbt"
    if (rbt.repOk()) print(rbt);
  }
  ...
}
```

Figure 2. Instrumented code for implementing Korat on JPF

There are two key differences that make our implementation more efficient than lazy initialization for enumerating desired object graphs. First, we pre-allocate objects that are re-used during exploration; in contrast, lazy initialization re-allocates objects every time an object field is initialized to a value that is not in the structure constructed thus far. Second, we put a priori bounds on the number of objects for each type (by setting the domains for each field); in contrast, lazy initialization instead bounds the length of pro-

gram paths explored by JPF. For enumerating desired object graphs, directly bounding the number of objects (as we do) provides a tighter bound on the search space. Thus, for testing programs, our implementation is more suitable for black-box testing, whereas lazy initialization is more suitable for white-box testing (e.g., to cover a path), where it may not be simple to a priori calculate tight bounds on the number of objects required. Our modifications to JPF further improve on this basic Korat implementation based on code instrumentation.

4. Optimizing search in JPF

We next describe the changes we made to JPF to speed up the Korat search. We group our changes based on the functionality of JPF that they target. The first group modifies the JVM interpreter in JPF to eliminate some code instrumentation discussed in Section 3; these changes by themselves do not speed up JPF significantly, but they make it easier to use JPF for Korat search as we do not have to manually instrument the code (or develop automatic instrumentation). The second group of changes targets state comparison, the most expensive part of the basic search in JPF. The third group targets state backtracking for heap, another expensive part that affects not only the backtracking time itself but also the bytecode execution time in JPF. The fourth group directly targets bytecode execution. The fifth group targets backtracking and execution for stacks and threads. The sixth group targets the general overhead costs of JPF. While each of these groups provides some speedup by itself, together they have a synergistic effect and provide more speedup than the sum of their individual speedups.

4.1. Modifying the interpreter

Korat search effectively requires a non-standard execution of bytecodes. Specifically, accesses to certain fields require execution as shown in the getter methods in the instrumented code in Figure 2 (checking shadow fields and potentially invoking `getInt`). However, code instrumentation is not the only way to implement a non-standard semantics. Another way is to change the interpreter/runtime itself; JPF was previously changed to implement non-standard semantics, e.g., for symbolic execution [3,9], but was not previously changed for Korat search [20, 33].

We have modified the JPF's interpreter to perform non-standard execution for search. Recall that JPF implements a (backtrackable) JVM. The implementation uses the Interpreter Design Pattern [15] that represents each bytecode with an object of some class. Specifically, the class `GETFIELD` implements the bytecode that reads a field from an object. We modified this class to implement the getter methods shown in Figure 2, namely to check if each appropriate field is initialized, and if not to create a non-deterministic choice point to initialize the field. This means that we do not need to instrument the code (by adding getter methods and replacing field accesses with calls to those methods) to have JPF perform the search.

An important question is for which fields to check whether they are initialized. With code instrumentation, this is determined *statically* by replacing some field accesses with getter methods. With modified interpreter, however, this is determined *dynamically* as the execution of finitization sets appropriate flags to mark that a field participates in the finitization. The latter is preferred since the former could replace (1) too many fields (thus unnecessarily slowing down the execution) or (2) too few fields (thus resulting in an incorrect execution unless a new instrumentation is performed to replace more field accesses). Therefore, modifying interpreter makes it much easier to use Korat on JPF. Moreover, it also provides a small speedup as shown in Section 5, mostly because JPF has fewer bytecodes to execute with a modified interpreter than with added instrumentation.

4.2. Reducing state comparison costs

JPF is a stateful model checker: it stores (hash values of) the states that it visits during a state-space exploration and compares (hash values of) newly encountered states with the visited states. In general, this comparison speeds up JPF as it avoids exploring the same portion of the state space multiple times. Our profiling showed that JPF can spend over 50% of overall time on operations related to state comparison (computing hash values, storing them, and actually comparing them).

Disable state comparison: An important but non-obvious insight is that Korat search does not need any state comparison. Namely, the search always produces different states because each non-deterministic choice point (either in the getter method with code instrumentation or in the field read with modified interpreter) assigns a new value to some field. Since all these fields are in the objects reachable from the root (the `RedBlackTree` object in our example from Figure 2), the states always differ. Thus, state comparison cannot find a state that has been already visited, and it is beneficial to turn state comparison off. It is easy to do so as JPF provides a configuration option for state comparison.

Reduce GC: Garbage collection (GC) is another operation in JPF related to state comparison. JPF performs GC before each non-deterministic choice point (call to `getInt`) both to reduce the required amount of memory and to improve the state comparison. GC helps the comparison as it avoids comparing the values of unreachable objects and can also help with comparing states based on isomorphism [18, 19, 21, 32]. However, since our search does not need state comparison, it is unnecessary to perform GC that

often. Note that we still need to perform GC occasionally for its other purpose, to reduce the amount of memory. We modified JPF to perform GC only when the amount of free memory for JPF gets under 10% of the total available memory. (The system calls `java.lang.Runtime.{free, total}Memory` return the amount of free memory and the total amount of memory in the JVM.)

4.3. Reducing backtracking costs for heap

State backtracking is another expensive operation in JPF. It enables JPF to explore various choices from non-deterministic choice points. JPF implements backtracking by storing and restoring the entire JVM state at the choice points [21]. JPF improves on naive storing and restoring by using state collapsing and sharing the substates between various states [1, 21, 32]. This significantly saves memory and makes it faster to store, restore, and compare states.

Undo heap backtracking: We modified JPF to restore states by keeping "deltas", i.e., by saving and undoing the state modifications between choice points. This is a known technique for manipulating states in explicit-state model checking [7]. Lerda and Visser [21] did not use this technique while initially developing JPF, as they (correctly) argued that the technique could slow down state comparison. Additionally, this technique could be slower than full storing/restoring when the state is relatively small at choice points but the execution between them is relatively long and has many state updates (e.g., creating temporary objects that get garbage collected). However, Korat search does not require state comparison at all and has relatively short execution between choice points. Therefore, it is quite beneficial to keep track of "deltas" rather than the entire states. Our "Undo" extension was included in JPF and is publicly available from the JPF web site [1]. It is also listed in a tool paper [17] as one of our four state extensions for JPF.

Singleton non-deterministic choice: We made another change related to backtracking, specifically in the JPF library for non-deterministic choice. Recall that `Verify.getInt(int lo, int hi)` returns a value between `lo` and `hi`. Our change is trivial but important: if `lo` and `hi` have the same value, then `getInt` need not create a backtracking point but should only return the value. The library did not check for this special case since most `getInt` calls are *manually* added to the program and would not even be used when only one value is needed. However, our search *automatically* executes Korat finitizations which use `getInt` to set the fields, e.g., to set the size of a data structure between some minimum and maximum size, where experiments usually require structures of only one given size. Our `getInt` change was also included in JPF [1].

4.4. Reducing bytecode execution costs

At its core, JPF is an interpreter for Java bytecodes. For some bytecodes that manipulate state or affect thread interleavings, JPF needs to perform customized operations (to enable backtracking) different from a regular JVM, but for most bytecodes, JPF simply performs as a regular JVM. However, being an interpreter, JPF is relatively slow in executing even regular bytecodes. In particular, we determined that while executing `repOk` methods such as the one shown in Figure 1, JPF can spend a lot of time executing library methods on collections such as sets, queues, or stacks.

Simple collections: An important characteristic of `repOk` methods is that they build fairly small collections. For example, the set `visited` in Figure 1 cannot have more than N elements, where N is the (small) bound for the number of nodes, given in the finitizations. We developed a library of collections that execute faster on JPF than the standard collections from the Java library (in the `java.util` package) when collections are small. The standard library is designed for large collections and uses algorithms with good asymptotic behavior but not necessarily good constants. For example, `java.util.TreeSet` implements sets with red-black trees that can insert, delete, and search for an element in $O(\log n)$ time but are fairly expensive for small sets. Our collections use simple, array-based implementations that take $O(n)$ for typical operations but perform much better for small collections that `repOk` methods build.

Execution on JVM: We additionally sped up the operations on our library structures by moving their execution from the JPF level to the underlying JVM level. Recall that JPF works as a special JVM executing on top of a regular, host JVM. Instead of executing operations by interpreting their bytecodes with the JPF's interpreter, our modified library executes the bytecodes directly on the host JVM. Note that this execution still needs to manipulate JPF state, e.g., if the execution adds an element to some set, JPF needs to be aware of this to properly restore the set when backtracking. We previously proposed a technique for executing JPF operations at the JVM level [10]. However, that technique requires translations between JPF and JVM states: first the JPF state is translated to the JVM level prior to the operation (or on-demand, during the operation) and then the JVM state is translated back to the JPF level at the end of the operation. The previous technique has a relatively big overhead for array-based collections and requires partially manual instrumentation for such collections. In contrast, our novel technique *executes bytecodes on the JVM level but directly manipulates the JPF state*. We had to manually write the library once (adding a collection when needed for the `repOk` of some data structure in our experiments), but it can be fully automatically executed (and the same collection reused for several data structures).

4.5. Reducing costs for stacks and threads

After all the previous changes, a major portion of JPF time goes on manipulation of stacks and threads. We developed several changes that reduce both backtracking and execution costs associated with stacks and threads.

Stack optimizations: JPF represents a stack for each thread as a vector of stack frames, which allows faster and more compact storing of states for backtracking by sharing the stack frames from various states [21]. However, this representation requires that stack frames be cloned—JPF uses "copy on write"—so that execution of one path does not affect stack frame stored for another path. Each stack frame contains several arrays that store values of method local variables and operand stack and track whether these are references or not (for GC). We implemented three optimizations for stack frames. First, we simplify stack frames by merging some arrays and representing reference arrays with bit vectors for the common case when there are fewer than 32 local variables. This change makes it faster to create, store, and clone stack frames; also, such simpler stack frames require less memory. (The next version of JPF will include similar changes developed independently of ours.) Second, we mark some stack frames to use shallow cloning rather than deep cloning, i.e., copying stack frames such that they share the arrays for local variables, which reduces JPF's execution time and memory requirements. This can be done for each choice point after which no live variable in a stack frame needs to be backtracked, e.g., in the `main` method in Figure 1 the computation after a return from `repOk` depends on the return value, and variable `bst` need not be backtracked since it cannot modified. We currently mark only `main` methods for all subject program, but a static or dynamic analysis could find more opportunities [12]. Third, we added code to completely avoid cloning stack frames that are not shared but execute the last choice for some choice point. Effectively, this avoids "copy on write" for a stack frame that is not shared (which our code tracks dynamically) and that will not be reused in the future, e.g., when the `Verify.getInt` from Figure 2 takes the last possible value.

Thread optimizations: JPF is a general model checker that supports multi-threaded code. Indeed, it is one of the strengths of JPF that it can be explore various interleavings of multiple threads. However, Korat operates on single-threaded code, and it is unnecessary to pay the overhead for maintaining some information required only for multi-threaded code. Specifically, we made two changes to JPF. First, at the end of every execution path, JPF terminates the main thread and removes it from the `ThreadGroup` object containing all non-terminated threads, which requires some synchronization. Single-threaded code does not use this object, so we added a JPF flag to ignore this object. Second, JPF uses thread information to assign identifiers to allocated objects (with the `NEW` bytecode) to achieve heap symmetry (since the order of concurrent allocations should not result in different heap states). We simplified the assignment of object identifiers for single-threaded code. (The next version of JPF will include a different scheme for identifiers.)

4.6. Reducing other overhead costs

JPF is built as a customizable and extensible tool [1]. Indeed, the primary design goal of JPF was not speed of exploration but ease of experimenting with novel exploration techniques. For example, JPF uses the Listener Design Pattern [15] to enable external listeners that monitor or alter state-space exploration. Additionally, JPF logs the executed bytecodes to help in debugging or replay. However, listeners and logging provide overhead in execution even when they are not needed, as for our search. Therefore, we disabled listeners and logging.

5. Evaluation

We next present an evaluation of the modifications we made to JPF while implementing the Korat search. We first describe the subjects used in our evaluation. We then present the speedup obtained by applying the modifications presented in Section 4 on the instrumentation-based version of Korat-JPF presented in Section 3. The key metric in our experiments is the time that JPF takes to perform the search for a given data structure and bound. We performed all experiments using Sun's JVM 1.5.0_13-b05 on a Pentium M 1.73GHz processor running Ubuntu Linux 7.04. Each run was limited to 1GB of memory. Finally, we present the impact our modifications have on memory usage.

5.1. Subjects

We evaluated the effect of our modifications using ten different data structures, previously used in other studies on testing and model checking, including generation of object graphs [6, 24]: `BinaryTree` is a simple binary tree implementation; `BinHeap` is an implementation of priority queues using binomial heaps [8]; `DisjSet` is an array-based implementation of a fast union-find data structure [8]; `DoublyLL` is an implementation of a circular, doubly-linked list (used for example in the Java Collections Framework, where this subject is representative for related linked data structures such as stacks and queues); `FaultTree` is an implementation of an (abstract) fault tree used to model system failures [30]; `HeapArray` is an array-based priority queue implementation (this subject is representative for array-based data structures such as stacks and queues or `java.util.Vector`); `RedBlackTree` is our running example

Experiment		Candidates	Structures
Subject	N	Explored	Generated
BinaryTree	11	3,162,018	58,786
BinHeap	8	1,323,194	603,744
DisjSet	5	413,855	41,546
DoublyLL	11	3,535,294	3,535,027
FaultTree	6	6,152,447	1,026,944
HeapArray	8	5,231,385	1,005,075
RedBlackTree	9	1,510,006	122
SearchTree	8	2,606,968	1,430
SinglyLL	11	10,639,556	678,570
SortedList	9	437,782	48,620

Figure 3. Exploration characteristics

from sections 2 and 3; SearchTree is a binary search tree implementation; SinglyLL is an implementation of a singly-linked list; and SortedList is similar to DoublyLL, with the additional requirement that the elements be sorted.

Figure 3 shows the data structures and characteristics of our experiments. For each subject, we instruct Korat-JPF to generate data structures of size exactly N; for example, RedBlackTree 9 instructs Korat-JPF to generate all valid red-black trees with 9 nodes. We tabulate the number of candidates that the search explores during the generation (i.e., the number of backtracks) and the number of valid, non-isomorphic structures that it generates. Notice that these experiments cover a diverse range of linked and array-based data structures, with the number of candidates ranging from several hundred thousand (DisjSet and SortedList) to ten million (SinglyLL), and the number of generated structures ranging from just over one hundred (RedBlackTree) to a few million (DoublyLL).

5.2. Speed up

Figure 4 shows the cumulative effect of applying our modifications to the base implementation of Korat-JPF. The first two columns indicate the data structure and bound for which generation was performed. The subsequent columns show the time required to generate all the valid structures for a given set of modifications.

The column *Base* contains running times for the basic instrumented version of Korat-JPF (Section 3). The column *Mode 1* shows the running times obtained by switching from the Korat implementation based on code instrumentation to the implementation based on modifying JPF's JVM interpreter (Section 4.1). Recall that the main goal of this change is to make it easier to use Korat-JPF rather than to improve the running time. The change still provides a small reduction of the time (under 10%) for some experiments but also provides a small increase of the time for three experiments (DisjSet, HeapArray, and RedBlackTree). The remaining columns show the running times for different sets of optimizations, applied cumulatively, i.e., *Mode* M includes all of the optimizations used in *Mode* $M-1$:

- *Mode 2* adds modifications that reduce state comparison costs (Section 4.2)
- *Mode 3* adds modifications that reduce backtracking costs for heap (Section 4.3)
- *Mode 4* adds modifications that reduce bytecode execution costs (Section 4.4)
- *Mode 5* adds modifications that reduce backtracking and execution costs for stacks and threads (Section 4.5)
- *Mode 6* adds modifications that reduce other overhead costs (Section 4.6).

The *Speedup* column shows the overall improvement obtained by applying all optimizations, i.e., the ratio of the times for *Base* and *Mode 6*. The speedup ranges from 10.58x (FaultTree) to 31.29x (BinaryTree), with a median of 16.18x. In all experiments, our modifications provide *an order of magnitude speedup*. In general, the speedup is larger for linked data structures than for array-based data structures; the main exception is the speedup for RedBlackTree being smaller than the speedup for HeapArray.

All modes contribute to the overall speedup. *Mode 2* contributes the most; recall that this is based on the insight that Korat-JPF does not require any state comparison. *Mode 3* also contributes significantly for all subjects; this is mainly due to our novel implementation of state backtracking in JPF (Section 4.3). *Mode 4* contributes significantly for subjects whose repOk methods use collections that are targeted by the modifications to reduce bytecode execution (Section 4.4). For the two subjects that do not use these collections, DisjSet and HeapArray, the running times should be the same for both *Mode 3* and *Mode 4*. The minimal speedup (for DisjSet) and slowdown (for HeapArray) shown in Figure 4 are due to the small imprecision in our time measurement, as opposed to any real change in performance. *Mode 5* contributes to exploration speedup by reducing both backtracking and execution costs for stacks and threads. *Mode 6* contributes the least.

5.2.1 Analysis of Mode 6

Our inspection of the number of operations (such as backtracks, stack frame clones, etc.) and the timing profile for Mode 6 shows that it is a near optimal solution for the current design of JPF. It would be too naive to claim that it is *the* optimal solution, but the inspection shows that the

Experiment		Time (sec)							Time
Subject	N	Base	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6	Speedup
BinaryTree	11	1,403.75	1,398.66	658.92	223.12	72.15	52.12	44.86	31.29x
BinHeap	8	885.94	866.78	551.16	278.70	90.11	81.90	66.89	13.25x
DisjSet	5	159.25	161.91	66.14	16.12	16.02	14.87	12.13	13.12x
DoublyLL	11	1,632.86	1,576.30	649.86	158.17	123.14	101.04	85.91	19.01x
FaultTree	6	5,348.47	5,102.74	3,456.12	2,405.72	631.90	634.98	505.57	10.58x
HeapArray	8	1,504.12	1,659.79	678.03	136.15	136.28	103.51	86.43	17.40x
RedBlackTree	9	886.79	890.70	365.47	128.05	112.23	101.05	77.44	11.45x
SearchTree	8	1,068.02	1,056.18	457.72	114.16	110.15	87.07	71.46	14.95x
SinglyLL	11	4,993.96	4,898.96	2,253.04	766.14	316.87	239.23	205.31	24.32x
SortedList	9	172.11	168.21	65.32	11.21	11.09	8.26	7.50	22.96x

Figure 4. Exploration time for various modifications to basic Korat-JPF

number of operations is the smallest possible, and the profile shows that most of the time goes into bytecode execution (with some time going to backtracking), evenly spread among the bytecodes with no apparent bottleneck. Since JPF is a bytecode interpreter, we expect that a further big speed up could be obtained by using a compiler instead.

5.2.2 Comparison with Korat on JVM

We also compared our most optimized Korat-JPF with the publicly available Korat implementation running on JVM [2]. The latter implements a customized backtracking using re-execution [6, 24], and we refer to it as Korat-JVM. For the bounds shown in Figure 4, the most optimized Korat-JPF is still 2.98x-12.27x (with a median of 7.62x) slower than Korat-JVM. However, one should consider this in the context of the overhead that bytecode execution has in JPF. We ran Korat-JVM on the JPF virtual machine, and the slowdown compared to the regular JVM can be huge, e.g., for `BinaryTree` and size 6, Korat-JVM on JPF took 29.29x more time than on JVM, and for size 11, Korat-JVM on JPF did not even finish in 4 hours.

5.3. Memory savings

Figure 5 shows the cumulative effect that applying our modifications to the base implementation of Korat-JPF has on memory usage. We used Sun's `jstat` monitoring tool [31] to measure the peak usage of garbage-collected heap in the JVM while running experiments for each mode. This measurement represents the most relevant portion of memory used during generation. The information is organized in a fashion similar to Figure 4. For lack of space and because results do not vary much with modes, we only show data for three of the seven execution modes.

The column *Base* shows the peak memory used by our basic instrumented version of Korat-JPF. *Mode 1* takes roughly the same amount of memory as *Base*. The column *Mode 2* shows the peak memory used when we applied the modifications designed to reduce state comparison costs. At this point, most of the variable memory allocation associated with the exploration has been eliminated. The column *Mode 6* shows the peak memory used when we applied all our modifications. The modes after *Mode 2* are focused on speeding up the exploration (as shown in Figure 4) and although they have a positive impact on memory usage (as indicated by by peak memory for *Mode 6* being smaller than that for *Mode 2*), the changes in *Mode 2* provide the vast majority of memory savings.

Experiment		Peak Memory (MB)			Memory
Subject	N	Base	M. 2	M. 6	Savings
BinaryTree	11	88.19	6.76	5.53	15.93x
BinHeap	8	35.32	6.68	5.67	6.23x
DisjSet	5	15.15	5.68	5.62	2.69x
DoublyLL	11	83.70	6.53	5.78	14.47x
FaultTree	6	258.91	7.07	6.20	41.75x
HeapArray	8	21.17	5.68	5.58	3.79x
RedBlackTree	9	63.52	7.00	5.95	10.66x
SearchTree	8	84.85	6.76	5.79	14.64x
SinglyLL	11	262.48	6.50	5.76	45.53x
SortedList	9	14.82	6.08	5.21	2.84x

Figure 5. Peak memory for selected modifications to basic Korat-JPF

6. Conclusion

We presented our use of an off-the-shelf model checker as an enabling technology to implement a constraint-based test generation approach. Specifically, we use JPF, a popular model checker for Java programs, to implement the

Korat search algorithm. A basic implementation of Korat in JPF using code instrumentation results in an unnecessarily slow search. We modify several core operations of JPF to speed up the search. Experiments using various subject data structures show that our modifications reduce the search time in JPF by over an order of magnitude. Moreover, our inspection of the number of operations (such as backtracks) shows that our solution is near optimal for the current design of JPF: JPF is a bytecode interpreter, so a big speed up could be obtained by using a compiler instead.

Acknowledgments. We thank Peter Mehlitz, Corina Pasareanu, and Willem Visser for help with JPF; Marcelo d'Amorim for discussions on JPF; and Aleksandar Milicevic and Sasa Misailovic for comments on our work. This material is based upon work partially supported by NSF Grant Nos. CCF-0746856, CCF-0702680, CNS-0615372, CNS-0613665, and IIS-0438967. Milos and Tihomir did a part of this work as undergraduate summer interns at the Information Trust Institute at the University of Illinois.

References

[1] Java PathFinder webpage. `http://javapathfinder.sourceforge.net`.

[2] Korat webpage. `http://korat.sourceforge.net`.

[3] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, pages 134–138, 2007.

[4] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[5] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *SAS*, 2006.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

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

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

[9] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE*, pages 59–68, 2006.

[10] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, volume 4260, pages 549–567, 2006.

[11] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *OOPSLA*, 2006.

[12] P. de la Cámara, M. del Mar Gallardo, and P. Merino. Abstract matching for software model checking. In *SPIN*, pages 182–200, 2006.

[13] B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.

[14] B. Elkarablieh, D. Marinov, and S. Khurshid. Efficient solving of structural constraints. In *ISSTA*, 2008.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, 1997.

[17] T. Gvero, M. Gligoric, S. Lauterburg, M. d'Amorim, D. Marinov, and S. Khurshid. State extensions for Java PathFinder. In *ICSE Demo*, 2008.

[18] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, pages 254–261, 2001.

[19] R. Iosif and R. Sisto. Using garbage collection in model checking. In *SPIN*, pages 20–33, 2000.

[20] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[21] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN*, pages 80–102, 2001.

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

[23] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[24] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 2005.

[25] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.

[26] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *ESEC/FSE*, 2007.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[28] R. Rugina. Quantitative shape analysis. In *SAS*, pages 228–245, 2004.

[29] K. Stobie. Model based testing in practice at Microsoft. *Electr. Notes Theor. Comput. Sci.*, 111:5–12, 2005.

[30] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, pages 133–142, 2004.

[31] Sun Microsystems. *jstat: Java Virtual Machine Statistics Monitoring Tool*. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html`.

[32] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[33] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, 2004.

[34] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *ASE*, pages 414–417, 2005.

[35] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, 2006.

[36] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of Java software using context-sensitive capture and replay. In *ESEC/FSE*, pages 85–94, 2007.

[37] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.