Optimizing Generation of Object Graphs in Java PathFinder

Milos Gligoric, Tihomir Gvero, Steven Lauterburg, Darko Marinov, Sarfraz Khurshid

JPF Workshop 1.5.8
Bugs–Six Decades Ago

1947: Harvard Mark II

Photo: National Museum of American History
Mars Polar Lander, 1999
Crashed–premature shut down at 40 meters altitude
USS Yorktown, 1997
“Dead in the water” for 3 hours

Photo: navsource.org
Java PathFinder

Java PathFinder (JPF) is a popular explicit-state model checker

- Directly checks properties of a wide range of Java programs

Implements a customized Java Virtual Machine

- Runs on the top of the host JVM

Two key operations for explicit state-space search:

- **State backtracking**: stores/restores state to backtrack the execution during the state-space exploration
- Control over non-deterministic choices (including thread scheduling)

Traditional focus: check properties of control, such as deadlock

- Recent work: also check properties of data, such as acyclicity
Structurally Complex Data

size: 9

```
[city = washington, [building = whitehouse, [wing = west, [room = oval-office]])]
[service = camera, [data-type = picture, [format = jpg]]]
[resolution = 640x480]
[accessibility = public]
```
Korat Framework

Enables systematic, constraint-based testing

• E.g., testing on all “small” inputs

Implements a constraint solver for **imperative predicates**

Takes two inputs:

• A Java predicate that encodes constraints that represent properties of desired object graphs
• A **bound** on the size of the graph

Generates all **valid** graphs (within the given bound)

Performs a **backtracking search**

• Systematically explores the bounded space of object graphs
Korat in JPF

Korat solver was originally developed from scratch

- However, it can leverage tools, such as model checkers, which support backtracking as a fundamental operation

JPF has been used to generate object graphs

We use JPF as an implementation engine for Korat

- Start with a simple instrumentation of the Java predicate
- Observe that a direct implementation makes generation of object graphs unnecessarily slow in JPF
- Explore changes (optimizations) that could speed-up Korat in JPF
JPF State Operations

JPF uses a special representation for the state of program it checks.

It performs three basic kinds of operations on the state representation:

- **Bytecode execution**: manipulates the state to execute the program bytecodes
- **State backtracking**: stores/restores state to backtrack the execution during the state-space exploration
- **State comparison**: detects cycles in the state space

Our changes target each of these operations.
Outline

Overview

Example

Korat search in JPF
Optimizing Korat search in JPF
Evaluation
Conclusions
Example Korat Input
A set implemented as a red-black tree

```java
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 = Factory.<Node>createSet();
        visited.add(root);
        Queue<Node> workList = Factory.<Node>createQueue();
        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
    }

    ...
}
```

Korat-JPF
Korat Input and Output

Input:

- **Method repOk**—checks whether an object graph indeed represents a valid red-black tree
- **Finitization**—provides:
  - A bound for the number of objects of each class in one graph, e.g., number $N$ of Node objects
  - A set of values for each field of those objects
    - E.g., root, left, right, and parent are either null or point to one of the $N$ Node objects

Output:

- Enumeration of all valid non-isomorphic object graphs, within the bounds given in the finitization
  - E.g., all valid red-black trees within the bounds
Korat Search

Systematically explores the bounded input space
• Orders values for each field to build its **field domain**
• Starts the search using candidate with all fields set to the first value in their domain
• Executes repOk on the candidate
  • Monitors repOk’s execution dynamically
  • Records the field access order according to first access
• Generates the next candidate based on the execution
  • Backtracks on the last field in field access order
    • Prunes the search
    • Avoids equivalent candidates
Uses bytecode instrumentation to insert monitors
Korat Search in JPF

Simple implementation of Korat using code instrumentation

- Use `Verify.getInt`, which returns a non-deterministic value, to index into a field domain
- Use shadow boolean fields to monitor field accesses
- Track assigned objects of a field domain to ensure exploration of non-isomorphic structures
Example: Code instrumentation in JPF

```java
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 {
        ...
    }

    boolean repOk() {
        /* same as before*/
    }

    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
    }
}
```
Example: Finitization in JPF

// "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 one call to "repOk" will backtrack a number of times,
    // setting the fields of objects reachable from "rbt"
    if (rbt.repOk()) print(rbt);
}
... // end of class RedBlackTree
}
Optimizing Search in JPF

Groups of changes made in JPF:

• Modifying interpreter
• Reducing state comparison costs
• Reducing backtracking costs for heap
• Reducing bytecode execution costs
• Reducing costs for stacks and threads
• Reducing other overhead costs

*These changes reduce the search time by over an order of magnitude*
Modifying Interpreter

**Idea:** instead of code instrumentation, change the interpreter itself

Modified the class that implements the bytecode that reads a field from an object

- To check whether a field is initialized
- If not to create a non-deterministic choice point

No need to instrument the code

Makes it much easier to use Korat on JPF
Reducing State Comparison Costs

JPF is a stateful model checker:

- Stores (hash value of) the states that it visits
- Compares (hash value of) newly encountered states with the visited states

**Idea**: disable state comparison

- Korat search does not need any state comparison
- The search always produces different states
Reducing State Comparison Costs (2)

**Idea:** reduce garbage collection (GC)

- GC helps the state comparison
- Unnecessary to perform GC that often
- Perform GC occasionally, only to reduce the amount of memory
Reducing Backtracking Costs for Heap

State backtracking is expensive operation

- Storing and restoring the entire JVM states at the choice points

**Idea:** undo backtracking

- Incrementally stores and restores states
- Only keeps track of the state changes that happen between choice points
- Later restores the state by undoing these state changes
Undo Backtracking

An execution of a field assignment

root.right = null

execution: remember change

<table>
<thead>
<tr>
<th>n2.left</th>
<th>n2.right</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>n1</td>
</tr>
<tr>
<td></td>
<td>n3</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n2.left</th>
<th>n2.right</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>n1</td>
</tr>
<tr>
<td></td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Undo Backtracking (2)

Reduces the execution time as it does not require JPF to clone the integer array that encodes all fields of an object whose one field is being written to.

```
root.right = null
```

**Execution:** remember change

**Backtracking:** undo change
Reducing Backtracking Costs for Heap (2)

**Idea:** special handling of singleton non-deterministic choice

- Verify.getInt(int lo, int hi) does not need to create a backtracking point if lo == hi

Both Undo backtracking and Singleton non-deterministic choice changes have been added to JPF and are publicly available
Reducing Bytecode Execution Costs

JPF is an interpreter for Java bytecodes
For most bytecodes, JPF simply performs as a regular JVM
Slow in executing even regular bytecodes

Idea: use simple collections
  • repOk methods build fairly small collections
  • A simple library of collections that execute faster on JPF, when collections are small

Idea: execute on JVM
  • Moving our library structures execution from the JPF level to the underlying JVM level
Reducing Costs for Stacks and Threads

A major portion of JPF time goes on manipulation of stacks and threads

**Idea**: optimize stack handling
- Simplify stack frames
- Shallow cloning rather than deep cloning of some stack frames
- Avoids “copy on write” when a stack frame is not shared

**Idea**: optimize thread handling
- Korat operates on single-threaded code
- Unnecessary to pay the overhead for multi-threaded code
  - Recall DynamicAreaIndex triples that Peter mentioned
Reducing Other Overhead Costs

JPF is built as an extensible tool

JPF uses:
  • Listeners
  • Logging

Listeners and logging provide overhead in execution even when they are not needed

Configuration flag to turn off listeners and logging
## Evaluation: Time Speedup

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

### Exploration time for various modifications to basic Korat-JPF

- **Mode 1:** Modifying interpreter
- **Mode M:** includes all of the optimizations used in **Mode M-1**
- **Mode 2:** Reducing state comparison costs
- **Mode 3:** Reducing backtracking costs for heap
- **Mode 4:** Reducing bytecode execution costs
- **Mode 5:** Reducing costs for stacks and threads
- **Mode 6:** Reducing other overhead costs

---

**Modifications of JPF reduce the search time by over an order of magnitude**

---

28
### Evaluation: Memory Savings

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Peak Memory (MB)</th>
<th>Memory savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>N</td>
<td>Base</td>
</tr>
<tr>
<td>BinaryTree</td>
<td>11</td>
<td>88.19</td>
</tr>
<tr>
<td>BinHeap</td>
<td>8</td>
<td>35.32</td>
</tr>
<tr>
<td>DisjSet</td>
<td>5</td>
<td>15.15</td>
</tr>
<tr>
<td>DoublyLL</td>
<td>11</td>
<td>83.70</td>
</tr>
<tr>
<td>FaultTree</td>
<td>6</td>
<td>258.91</td>
</tr>
<tr>
<td>HeapArray</td>
<td>8</td>
<td>21.17</td>
</tr>
<tr>
<td>RedBlackTree</td>
<td>9</td>
<td>63.52</td>
</tr>
<tr>
<td>SearchTree</td>
<td>8</td>
<td>84.85</td>
</tr>
<tr>
<td>SinglyLL</td>
<td>11</td>
<td>262.48</td>
</tr>
<tr>
<td>SortedList</td>
<td>9</td>
<td>14.82</td>
</tr>
</tbody>
</table>

Peak memory for selected modifications to basic Korat-JPF

- Used Sun's jstat monitoring tool to measure the peak usage of garbage-collected heap
Summary

Implemented the Korat search algorithm in JPF
A basic implementation results in a slow search
Modified several core operations of JPF to speed up the search
Our modifications reduce the search time in JPF by over an order of magnitude
Two modifications are already included in the publicly available JPF
  • http://mir.cs.uiuc.edu/jpf

Future work
  • JPF as a solver for constraints in other languages
  • Parallelize JPF constraint solver