Test Input Generation for Java Containers using State Matching

Willem Visser
RIACS/NASA Ames
Moffett Field, CA 94035, USA
wvisser@email.arc.nasa.gov

Corina S. Păsăreanu
QSS/NASA Ames
Moffett Field, CA 94035, USA
pcorina@email.arc.nasa.gov

Radek Pelánek
Masaryk University
Brno, Czech Republic
xpelanek@fi.muni.cz

ABSTRACT

The popularity of object-oriented programming has led to the wide use of container libraries. It is important for the reliability of these containers that they are tested adequately. We describe techniques for automated test input generation of Java container classes. Test inputs are sequences of method calls from the container interface. The techniques rely on state matching to avoid generation of redundant tests. *Exhaustive techniques* use model checking with explicit or symbolic execution to explore *all* the possible test sequences up to predefined input sizes. *Lossy techniques* rely on abstraction mappings to compute and store abstract versions of the concrete states; they explore *under-approximations* of all the possible test sequences.

We have implemented the techniques on top of the Java PathFinder model checker and we evaluate them using four Java container classes. We compare state matching based techniques and random selection for generating test inputs, in terms of testing coverage. We consider basic block coverage and a form of predicate coverage - that measures whether all combinations of a predetermined set of predicates are covered at each basic block. The exhaustive techniques can easily obtain basic block coverage, but cannot obtain good predicate coverage before running out of memory. On the other hand, abstract matching turns out to be a powerful approach for generating test inputs to obtain high predicate coverage. Random selection performed well except on the examples that contained complex input spaces, where the lossy abstraction techniques performed better.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation, Verification

Copyright 2005 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ISSTA'06 July 17–20, 2006, Portland, Maine, USA.
Copyright 2006 ACM 1-59593-263-1/06/0007 ...$5.00.

Keywords

Software Model Checking, Unit Testing, Symbolic Execution, Abstraction, Random Testing

1. INTRODUCTION

Object oriented programming is fast becoming the paradigm of choice in everything from web applications to safety critical flight control software in the next generation of NASA manned missions. Modern object oriented languages typically come with libraries of container classes that are heavily reused without much concern given to the correctness of these container implementations. To ensure the reliability of systems built with such containers, they must be tested adequately. The large number of test inputs for thorough testing makes *automated* test input generation imperative.

This paper presents techniques for automated test input generation of container classes that use *state matching* to avoid generation of redundant tests. Test inputs are sequences of method calls from the container interface, that cover the relevant structural and behavioral aspects of the code. We use a model checker to *exhaustively* try all combinations of method calls and parameters to these calls up to a specified limit, but after each call the state of the container is examined to see if it can be "matched" with a previously stored state; if so, that sequence is discarded, if not the search continues with the next call.

During this search the testing coverage is measured and whenever new coverage is obtained the sequence of calls to achieve that coverage is recorded. We consider basic block coverage, as a representative example of simple structural coverage, and a form of predicate coverage [3] which measures the coverage of all the combinations of program predicates; predicate coverage is more difficult to achieve than basic block coverage.

The large amount of input data necessary to test the containers made us investigate an alternative technique, which uses symbolic, rather than explicit, execution, i.e. instead of concrete parameters to interface method calls it uses symbolic parameters. Symbolic execution [22] manipulates symbolic states, representing *sets* of concrete states, and *partitions* the input domain of the interface methods into non-overlapping subdomains, according to different paths that are taken during symbolic execution. Therefore, this technique has the potential to yield significant improvement over the "explicit" exhaustive technique. We describe a method for examining when a symbolic state is *subsumed* by another symbolic state. This is used for state matching to determine whether a test specific sequence can be discarded by the

model checker. Furthermore, we show that this approach scales better than the exhaustive explicit technique.

Even with state matching, the number of test sequences that needs to be explored with the explicit or symbolic techniques quickly becomes intractable – due to the state space explosion problem. We therefore define abstraction mappings to be used for state matching, to further reduce the state space explored by the model checker. More precisely, for each explored state, the model checker computes and stores an abstract version of the state, as specified by the abstraction. State matching is then used to determine if an abstract state is being re-visited. This technique is *lossy*, since parts of the feasible input sequences can be discarded due to abstraction. We introduce here a simple but powerful abstraction that records only the structure or *shape* of the container, while it discards the data stored in the container. We show that this lossy technique is the most effective with respect to coverage achieved.

We have implemented the techniques in a unified framework and we evaluate them on several container classes. The framework is built on top of the Java PathFinder [19, 28] model checker that already supports symbolic execution. Our framework also incorporates a technique based on random selection – and we use it as a point of comparison with the other techniques. As mentioned, we evaluate the techniques in terms of basic block and predicate coverage achieved by the generated test sequences. Predicate coverage is motivated by the observation that certain subtle program errors (that go undetected in the face of 100% basic block coverage) may be due to complex correlations between program predicates [3]. Therefore predicate coverage is a good addition to basic block coverage for evaluating test input generation strategies.

Although there is a lot of related work (presented in Section 6), we are not aware of a framework or a study that implements and compares explicit and symbolic techniques for test input generation with random selection in terms of structural coverage, let alone in terms of predicate coverage, as we do here.

The contributions of the paper are the following:

- Framework for test input generation for Java container classes. The framework incorporates explicit and symbolic techniques and uses state matching to avoid generation of redundant tests.

- Automated support for *shape abstraction* to be used during state matching. The abstraction can be used with both explicit and symbolic techniques. The abstraction is shown to be the most effective, as compared to all the other techniques.

- Evaluation of test generation approaches on four nontrivial Java container classes measuring the performance in achieving both a simple structural coverage and a form of predicate coverage. The evaluated approaches range from exhaustive testing, partition testing using symbolic execution and random testing.

2. BACKGROUND

We describe here the Java PathFinder (JPF) tool [19, 28] and its extension with a symbolic execution capability. Section 4 shows how to use JPF for test input generation.

2.1 Java PathFinder

JPF is an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). JPF handles all the Java language features and it also supports program annotations that are added to the programs through method calls to a special class `Verify`. We used the following methods from `Verify`:

`beginAtomic() ... endAtomic()` specify that the execution of the enclosed block should proceed atomically.

`random(n)` returns values $[0, n]$ nondeterministically.

`ignoreIf(cond)` forces the model checker to backtrack when `cond` evaluates to true.

By default, JPF stores all the explored states and it backtracks when it visits a previously explored state. Alternatively, the user can customize the search (by forcing the search to backtrack on user-specified conditions) and it can specify what part of the state (if any) to be stored and used for matching. We used these features to implement our test generation techniques that use model checking with abstract matching and random search. JPF also supports various search heuristics [13], that can be used to guide the model checker's search.

2.2 Symbolic Execution

Symbolic execution [22] allows one to analyze programs with un-initialized inputs. The main idea is to use *symbolic values*, instead of actual data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`) and a program counter. The path condition is a (quantifier free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path.

In previous work [21], we have extended JPF to perform symbolic execution for Java programs. The approach handles dynamically allocated data, arrays, and multi-threading. Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions. A Java implementation of the Omega library [25] is used to check satisfiability of numeric path conditions (for linear integer constraints).

3. TEST INPUT GENERATION

In this section we present our framework for generating test inputs for Java container classes. We illustrate our approach on a Java implementation of a binary search tree (see Figure 1). Each tree has a `root` node. Each node has an integer `elem` field and `left` and `right` children. Values are added and removed from the tree using the `add` and `remove` methods respectively.

A test input for `BinTree` consists of a sequence of method calls in the class interface (e.g. `add` and `remove`), with corresponding method arguments, that builds relevant object states and exercise the code in some desired fashion. Here is an example of a test input for `BinTree`:

```
class Node { ...
  public int elem;
  public Node left, right;
}
public class BinTree {
  private Node root;
  ...
  public void add(int x) { ... }
  public boolean remove(int x) { ... }
}
```

Figure 1: Java declaration of a binary tree

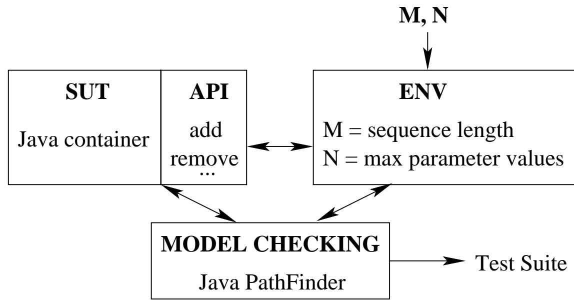

Figure 2: Test Generation Framework

```
BinTree t = new BinTree();
t.add(1); t.add(2); t.remove(1);
```

3.1 Framework

The framework for test input generation is illustrated in Figure 2. For each container (the system under test `SUT`) we built a *nondeterministic* environment `ENV`, i.e. a test driver that executes *all* sequences of API method calls up to a user-specified size `M`. JPF is used to enumerate all these sequences. The model checker's state matching capability avoids the exploration (and generation) of redundant tests.

The framework implements the following techniques (they are described in detail in the next section):

- exhaustive explicit execution with state matching
- explicit execution with abstract matching
- symbolic execution with subsumption checking
- symbolic execution with abstract matching
- random selection (no state matching).

For the techniques that use explicit execution or random selection, the user also needs to specify the range of values for the method parameters `[0, N-1]`. `N` is not needed when performing symbolic execution, since in this case the methods are executed with symbolic parameters.

3.2 Testing Coverage

The model checker analyzes the nondeterministic environment and it generates method sequences that achieve the desired testing coverage. We use basic block coverage, as a representative example of a widely used structural coverage measure. At each basic block, we also measure the coverage of all the combinations of a set of predicates chosen

```
static int M; /* sequence length */
static int N; /* parameter values */

static BinTree t = new BinTree();
public static void main(String[] args) {...
1:  for (int i=0;i<M;i++) {
2:    Verify.beginAtomic();
3:    int v = Verify.random(N-1);
4:    switch (Verify.random(1)) {
5:      case 0: t.add(v); break;
6:      case 1: t.remove(v); break;
      }
7:    Verify.endAtomic();
8: /* Verify.ignoreIf(store(abstractMap(t))); */
   }  }
```

Figure 3: Environment for explicit search

from conditions in the source code. We refer to the latter as *predicate coverage*, although it is strictly speaking only a simplified version of the predicate coverage as defined in [3] – where all predicates in the code are used rather than a subset as done here. For our purposes this is sufficient, since we are not measuring the adequacy of a test suite, but rather use it as a measure to compare test generation strategies.

The code of the methods is instrumented to record the coverage, e.g. basic block or predicate coverage. Whenever the model checker executes an uncovered block (or a new predicate combination), it outputs the current test sequence.

We should note that basic block coverage is a simple structural coverage whereas predicate coverage requires more behaviors (paths) to be followed through the code to obtain good coverage – this is in part due to the predicates coming from different portions of the code.

Also note that we have no way of computing the optimal predicate coverage for each SUT (hence we will refer to the highest observed coverage as the optimal). We use the rate of increase in the number of predicate combinations covered to evaluate whether a particular testing technique is helpful in covering paths not previously executed.

3.3 Testing Oracles

Method post-conditions can be used as test oracles to check the correctness of container methods. JPF also supports partial correctness properties given as assertions in the code and temporal logic specifications. In the experiments reported in this paper, we used JPF to check just absence of run-time errors, e.g. absence of uncaught exceptions.

4. TEST GENERATION TECHNIQUES

In this section we describe the techniques that are implemented in the test input generation framework, namely exhaustive explicit execution, explicit execution with abstract matching and symbolic execution (with subsumption checking and with abstract matching). We also describe how to use random selection for test input generation and we discuss the search order used in the model checker.

4.1 "Classical" Exhaustive Explicit Execution

We illustrate this technique using the `BinTree` example introduced in the previous section. The testing environment

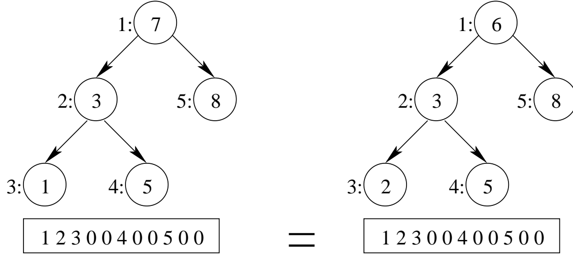

Figure 4: Abstraction recording shapes

is illustrated in Figure 3. The environment executes atomically *all* the sequences of `add` and `remove` methods up to the pre-specified sequence size `M`. The input values are chosen nondeterministically from range `[0, N-1]`. Line 8 is discussed in Section 4.2. As discussed, "classic" explicit state model checking is then used to search the state space of the program defined in Figure 3. The model checker's *default* state matching capability is used to avoid exploration and generation of redundant test sequences.

This straightforward approach does not scale well for large values of `M` and `N` – the number of possible test sequences becomes quickly intractable (the state space explosion problem). One way to address this problem is to use heuristic search; JPF supports several heuristics (guided search, beam search). Another solution is to perform explicit execution with abstract matching as described below.

4.2 Explicit Execution with Abstract Matching

The idea is to use the model checker to perform the explicit execution of all the possible method sequences (as above) but to store *abstract* versions of the explored program states, and use these abstract states to perform state matching (and to backtrack if an abstract state has been visited before). This effectively explores an under-approximation of the space of possible method executions.

In order to apply this technique for `BinTree` we use the environment illustrated in Figure 3, in which statement `8: Verify.ignoreIf(store(abstractMap(t)))` is included.

`abstractMap` computes an abstraction of the concrete container state of the binary tree referenced by `t`;

`store` directs the model checker to store the computed abstraction;

`Verify.ignoreIf` directs the model checker to backtrack if it has seen this abstraction before.

Note that state matching is now performed only on the state of the container object (referenced by `t`). This allows us to abstract away the information that is irrelevant to test generation, i.e. the values of local variables `i` and `v` are no longer considered to be part of the state.

JPF provides *automated support* for two powerful abstractions, that we have found useful in the analysis of containers.

- *The shape abstraction* records only the (concrete) heap shape of a container, while it abstracts away the data fields from each container element. The abstraction is illustrated in Figure 4, which depicts two binary search trees. Circles denote tree nodes; numbers inside circles denote the `elem` values; null nodes are not represented. The trees have the same heap shape - hence they will be matched during model checking (although the actual `elem` values are not the same). Heap shapes are represented in a normalized form, as sequences of integers (depicted in rectangles in Figure 4), and are obtained through a process called *linearization* [18, 32]. The linearization of an object (e.g. the tree root) starts from the root and traverses the heap in depth first search order; it assigns a unique identifier to each object and it backtracks when it detects a cycle; null pointers have values 0. Comparing shapes reduces to comparing sequences. Linearization has complexity $O(n)$ (where n is the number of heap nodes that are reachable from the root) and it can be performed efficiently during garbage collection.

- *The complete abstraction* records the shape of the container together with *all* the data fields from each container element. Strictly speaking, this is not really an abstraction (there is no information loss) but rather a canonical complete encoding of the container state, similar to the linearization used for representing the complete concrete heap (shape plus data), to achieve heap symmetry reduction in model checking [18].

```
static int M; /* sequence length */

static BinTree t = new BinTree();
public static void main(String[] args) {...
1:  for (int i=0;i<M;i++) {
2:    Verify.beginAtomic();
3:    SymbolicInt v = new SymbolicInt(''v''+i);
4:    switch (Verify.random(1)) {
5:      case 0: t.add(v); break;
6:      case 1: t.remove(v); break;
      }
7:    Verify.endAtomic();
8:    Verify.ignoreIf(checkSubsumptionAndStore(t));
    }  }
```

Figure 5: Environment for symbolic search

4.3 Symbolic Execution with State Matching

The test generation techniques that we have presented so far use concrete values for the method parameters. We now present an alternative technique, which assigns symbolic values for the input parameters, and it uses JPF to perform *symbolic* execution of the data structure's methods. The model checker manipulates symbolic states which describe *sets* of concrete states. As a result, this technique has the potential to yield significant improvements over explicit execution techniques.

To generates test inputs for `BinTree` (with symbolic execution) we use the environment illustrated in Figure 5. The environment is similar to what we had before, except now the type of variable v is `SymbolicInt` (rather than `int`) and `v` is assigned a new symbolic value each time the `for` loop is executed. Moreover, the code of the `add` and `remove` methods is instrumented to enable JPF to perform symbolic execution. State matching (line `8`) is discussed below.

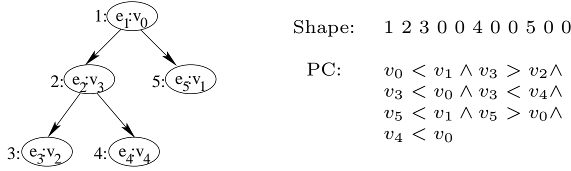

Figure 6: A symbolic state

Here is an example of a generated test sequence:

```
BinTree t = new BinTree();
t.add(v0);t.add(v1); t.remove(v2);

PC: v2 == v1 && v2 < v0 && v1 < v0;
Solution: v0: 1, v1: 0, v2: 0;
```

The path condition (PC) encodes the constraints on the input parameters. JPF also solves the constraints and it provides numeric solutions to be used as the concrete parameters for the actual test input. Our implementation is currently handling linear integer constraints. Other numeric domains could be handled similarly (provided the availability of appropriate decision procedures).

Let us now analyze a symbolic object state at line `8` - as illustrated in Figure 6. In each node, we write the symbolic value of the `elem` field, e.g. $e_1 : v_0$ means that the `elem` field of node 1 (e_1) has symbolic value v_0 ($v_0 ... v_5$ denote the symbolic values that were given as input parameters). The path condition encodes the constraints on the input values, and it may refer to symbolic values that are no longer stored in the tree, e.g. v_5. Intuitively, this means that this particular tree was created by a sequence that contained a `remove` call which removed value v_5 from the tree.

For state matching, we normalize the representation for symbolic states, using existential quantifier elimination. Intuitively, we are only interested in the relative order of the elements in the tree. For the example presented in Figure 6, we write the following constraints:

$$\exists v_0, v_1, v_2, v_3, v_4, v_5 :$$
$$e_1 = v_0 \wedge e_2 = v_3 \wedge e_3 = v_2 \wedge e_4 = v_4 \wedge e_5 = v_1 \wedge PC$$

We use the Omega library for existential quantifier elimination, which results in the following simplified constraints:

$$e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 > e_1 \wedge e_4 < e_1$$

The normalized symbolic state (shape plus simplified constraints) is used for state storing and comparison. A symbolic state encodes all the concrete states that have the same shape and whose elements satisfy the constraints.

Since symbolic states represent multiple concrete states, state matching involves checking *subsumption* between states. Let s_1 and s_2 be two symbolic states, and let $\gamma(s_1)$ and $\gamma(s_2)$ denote the sets of concrete states represented by s_1 and s_2 respectively. A symbolic state s_1 *subsumes* another symbolic state s_2, written $s_1 \supseteq s_2$, if the set of concrete states represented by s_1 contains the set of concrete states represented by s_2, i.e. $\gamma(s_1) \supseteq \gamma(s_2)$.

We check subsumption of two object states by checking that they have the same shape (as given by linearization) and that there is a valid implication between the corresponding constraints - we use the Omega library for checking validity. For example, in Figure 7, *Old state* $\supseteq$ *New state*: they have the same shape and the following implication is valid:

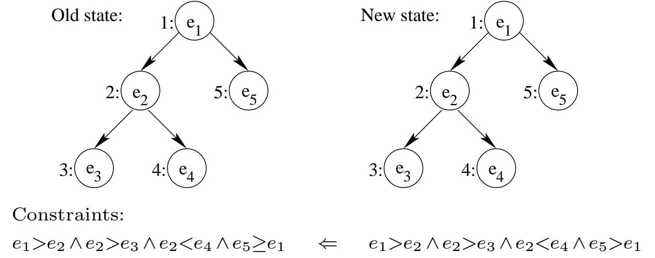

Figure 7: Two symbolic states

$$e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 \geq e_1$$
$$\Leftarrow e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 > e_1$$

In our framework, we have implemented bi-directional subsumption checking (line `8` in Figure 5). Let new_s denote a new symbolic object state, and let old_s denote a previously visited and stored symbolic state:

- If $old_s \supseteq new_s$, then `checkSubsumptionAndStore` returns `true` and the model checker backtracks.

- If $new_s \supseteq old_s$ then old_s is replaced with new_s (new_s is "more general" than old_s), `checkSubsumptionAndStore` returns `false` and the model checker's search continues (it does not backtrack).

Note that for each heap shape, we would like to store a *disjunction* of constraints, i.e. to store *unions* of symbolic states. In this case the bi-directional subsumption checking would no longer be needed. However, a small technicality prevents us (a bug in the Java implementation of the Omega library that JPF uses).

We should also note that symbolic execution can be used with abstract matching, i.e. replace line `8` in Figure 5 with `Verify.ignoreIf(store(abstractMap(t)))`. In particular, the model checker can use only the *shape* of a symbolic object state, for storing and matching, while discarding the numeric constraints (see the *shape abstraction* in the previous section).

4.4 Random Selection

The environment that we use for random selection is similar to the one presented in Figure 3, except that the nondeterminism is solved by random choice. When one (random) run is completed the search is restarted from the initial state and this process is repeated up to a user specified limit. In our experiments, we set the limit on the number of runs to 1000. Random search can be run stand-alone or using JPF – for our experiments we chose to run it inside of JPF. Note that due to technical reasons of JPF implementation, states are stored during the search (but they are never used).

4.5 Search Order

When considering an approach that uses state matching to prune the search space of test input sequences up to a *fixed* length we should note that we prefer to use breadth-first search order (BFS) rather than depth-first search (DFS) order. The reason is that DFS can miss portions of the state

space due to matching states that were created by a shorter sequence with those previously generated by a longer sequence (which was truncated due to hitting the length limit). This problem is amplified when considering abstract state matching, i.e. when matching states that are not necessarily identical. Therefore, all the test input generation techniques (besides random search) are used with BFS. Of course BFS also has the desirable characteristic that it produces shorter input sequences.

5. EVALUATION

As mentioned, we used the JPF model checking tool (version 3) to implement our testing framework. In particular, we used the *listener* mechanism [19] to observe the sequences of API calls performed and output the sequence when a specific coverage goal is reached. This test listener keeps track of the coverage obtained and calculates the average test input length. The user can specify on the command line the techniques to be used during the analysis. For the experiments we also added a facility to run tests described in a configuration file. The results of each run are collected in a file and a script generates a latex table with the results sorted by coverage.

5.1 Experimental Set-up

As a system under test we used Java implementations of four container classes: binary tree (`BinTree` – 154 LOC), binomial heap (`BinomialHeap` – 355 LOC), Fibonacci heap (`FibHeap` – 286 LOC), and red-black tree (extracted from `java.util.TreeMap` – 580 LOC). The methods of these classes were instrumented to measure basic block coverage (which implies statement coverage). At each basic block, we also measure predicate coverage. As mentioned, we have no way of computing the optimal predicate coverage. Therefore, we refer to the highest obtained coverage as optimal. Also note that our coverage numbers are absolute and not percentages as is commonly the case for test adequacy measures.

Note that we only focus here on obtaining code coverage and not on finding errors – this was a conscious decision to avoid bias from different fault seeding approaches. However in the future we would like to investigate whether the tests that obtain high coverage are also likely to detect faults.

Each container class is augmented with an environment as described in Section 4. For `BinTree` and `TreeMap` we only considered `add` and `remove` API calls. For `FibHeap` we also considered `removeMin` and for `BinomialHeap` we considered `add`, `remove`, `extractMin` and `decreaseKey(x,y)`. We considered these additional methods to determine the sensitivity of the techniques to the complexity of the environment.

We compare all the techniques described in the previous section. We divide the techniques into two categories: *exhaustive* and *lossy*. Exhaustive techniques include: explicit state model checking, explicit execution with complete abstract matching (i.e. linearization of a structure with all fields included) and symbolic execution with subsumption checking. Lossy techniques include: explicit and symbolic execution with abstract matching based only on shape and random selection.

For the techniques that use abstract matching, the order in which the state successors are generated can impact the search performance significantly. Therefore, we consider here successors taken in random order and we repeat each experiment 10 times. We run each technique for different values of sequence length `M` (from 1 to 30). For techniques which perform explicit execution we also need to specify the number of input parameters (`N`). In order to make the experiments tractable, we always set `M` = `N`. Note that this decision is quite justified in the context of containers, since the sequence length typically defines the size of the container, if each value added to the container is unique. If `M > N` then containers of size `M` cannot be generated. For random search we considered sequences up to length 50 and for each length we perform 1000 runs – as with the other lossy techniques each configuration is repeated 10 times. Since we use longer sequences for random it might be argued that `M = N` is unfair, and therefore we did some additional experiments where `M > N` when using random search (discussed later in the section).

5.2 Results

The results for exhaustive vs. lossy techniques measuring basic block and predicate coverage are reported in Tables 1–4. These results were produced from more that 10000 runs that took two months CPU time to complete. The results are split into four tables to show the difference in basic block coverage and predicate coverage during exhaustive and lossy search. The exhaustive experiments were preformed on a 2.66GHz Pentium machine running Linux and the lossy experiments on a 2.2Ghz Pentium running Windows 2000. In all cases memory was limited to 1GB.

For each technique we report the best result, i.e. the best coverage that was obtained at the shortest sequence length without running out of memory. Due to the randomization in the lossy techniques, it may happen that some results are obtained "luckily". We report only "stable" results, i.e. results achieved by all runs with a given parameter. The exception is random selection, where we report the best result, even if it happened only on one run. It turned out that the best results were always stable (of course excluding those for random selection), i.e. all 10 runs reported the same results.

We report the coverage, the sequence length (the minimum sequence length at which the coverage was obtained), the time taken (in seconds), memory used (in MB) and the average test sequence length. For the lossy techniques, the time, memory and average length are calculated by taking an average of the 10 runs. Numbers in bold show the maximum sequence length for which exhaustive results could be obtained.

5.3 Discussion

We will follow each discussion segment with some concrete conclusions (given in *italics*).

5.3.1 Exhaustive vs. lossy techniques

It is interesting to first note the complexity of some of the analyzed containers. For example, one needs sequences of length 14 to obtain basic block coverage (therefore also statement coverage) for `BinomialHeap` – 21 is the optimal coverage. From the exhaustive techniques only the symbolic execution approach using subsumption achieved this coverage. For `FibHeap` the optimal coverage is 25 and none of the exhaustive techniques could obtain this coverage before running out of memory. For `BinTree` and `TreeMap` the exhaustive techniques fared better and only model checking failed to get the optimal coverage for `TreeMap`. It is interesting to note that the two cases for which the exhaustive

Table 1: Exhaustive Techniques – Basic Block Coverage

Container	Technique	Coverage	Seq. Length	Time (s)	Memory (MB)	Avg. Length
BinTree	Model Checking	14	3	1	4	2.2
	Complete Abstraction	14	3	1	3	2.2
	Symbolic Subsumption	14	3	1	4	2.2
BinomialHeap	Model Checking	17	**5**	35	129	2.8
	Complete Abstraction	17	5	8	29	2.8
	Symbolic Subsumption	21	**14**	910	1016	4.2
FibHeap	Model Checking	20	**5**	55	214	3.7
	Complete Abstraction	24	7	8	19	4.2
	Symbolic Subsumption	24	7	15	54	4.2
TreeMap	Model Checking	37	**6**	38	243	4.2
	Complete Abstraction	39	7	9	34	4.3
	Symbolic Subsumption	39	7	15	22	4.3

Table 2: Lossy Techniques – Basic Block Coverage

Container	Technique	Coverage	Seq. Length	Time (s)	Memory (MB)	Avg. Length
BinTree	Shape Abstraction	14	4	1	3	2.2
	Symbolic Shape Abstraction	14	3	1	4	2.2
	Random Selection	14	3	7	3	2.4
BinomialHeap	Shape Abstraction	21	15	7	8	4.3
	Symbolic Shape Abstraction	21	14	1084	1016	4.2
	Random Selection	21	32	59	9	13.2
FibHeap	Shape Abstraction	25	12	26	34	4.4
	Symbolic Shape Abstraction	25	12	216	608	4.5
	Random Selection	25	25	41	9	10.2
TreeMap	Shape Abstraction	39	10	2	6	4.6
	Symbolic Shape Abstraction	39	7	7	22	4.3
	Random Selection	39	10	18	5	7.1

techniques fare better have simpler environments than the two cases where these techniques perform less well. All the lossy techniques achieved the optimal basic block coverage and where comparable, they achieved it faster and with less memory than the exhaustive techniques.

We anticipated that the exhaustive techniques would easily generate all tests to obtain basic block coverage – this is clearly not the case. The lossy techniques seem better suited for achieving code coverage. "Classic" model checking scales poorly even for the basic block coverage.

5.3.2 Symbolic execution

With the exception of `BinTree` – which is the simplest example – none of the exhaustive techniques obtain the optimal predicate coverage. With one exception, the symbolic execution with subsumption performs the best of the exhaustive techniques for both coverage measures. This is to be expected since the symbolic reasoning covers infinitely more cases than the explicit execution.

Symbolic execution with subsumption checking is, as anticipated, the most effective exhaustive technique. For the `TreeMap` *example it achieves good predicate coverage, even considering the lossy techniques.*

5.3.3 Abstract matching

State matching based on the shape abstraction is a lossy technique that performs the best of all the techniques: not only does it obtain the highest coverage (joint with others in some cases), but it obtains it for shorter sequences and it is faster. Only random selection, that essentially has no memory footprint, uses less memory when coverage is the same. We conjecture that for the analyzed containers, the shape is a very accurate representation of its state and hence the shape abstraction is appropriate here. It is an open question whether this will hold for general programs – it is likely to be the case for programs that manipulate complex data.

The complete abstraction that takes the shape and all fields into account performs almost as well, but uses more time and memory. This technique also performs consistently better than "classic" model checking which is closely related, but takes more than just the state of the container in consideration for state matching. Note that the complete abstraction also includes what is called a (data) symmetry reduction in model checking [18], and points to the fact that this kind of reduction is very useful in analyzing containers through API calls as we do here.

Shape abstraction as a means for state matching performs better than the other techniques considered. We conjecture that for the analyzed containers, the shape is a very accurate representation of its state. When doing test input generation for programs that manipulate complex data, this should be tried before other, more expensive, techniques.

5.3.4 Random selection

This is a traditional approach to test case generation; it is not based on state matching, hence it is the dual of the other methods suggested here and forms a useful point of comparison. Interestingly, in the related literature, it is almost never included in this kind of comparison. Here random search got

Table 3: Exhaustive Techniques – Predicate Coverage

Container	Technique	Coverage	Seq. Length	Time (s)	Memory (MB)	Avg. Length
BinTree	Model Checking	54	**6**	81	251	3.5
	Complete Abstraction	54	6	14	84	3.5
	Symbolic Subsumption	54	6	19	39	3.5
BinomialHeap	Model Checking	34	**5**	43	130	3.4
	Complete Abstraction	39	**6**	93	365	3.8
	Symbolic Subsumption	84	**14**	954	1016	6.8
FibHeap	Model Checking	31	**5**	59	208	4.0
	Complete Abstraction	89	**11**	733	1016	6.8
	Symbolic Subsumption	76	**9**	187	582	6.1
TreeMap	Model Checking	55	**6**	38	229	4.5
	Complete Abstraction	95	**10**	271	844	5.8
	Symbolic Subsumption	104	**12**	594	896	6.3

Table 4: Lossy Techniques – Predicate Coverage

Container	Technique	Coverage	Seq. Length	Time (s)	Memory (MB)	Avg. Length
BinTree	Shape Abstraction	54	9	4	11	3.6
	Symbolic Shape Abstraction	54	6	21	35	3.5
	Random Selection	54	8	15	4	6.2
BinomialHeap	Shape Abstraction	101	29	42	26	9.0
	Symbolic Shape Abstraction	84	14	1050	1016	6.8
	Random Selection	94	48	85	13	32.8
FibHeap	Shape Abstraction	93	15	243	292	7.1
	Symbolic Shape Abstraction	92	13	539	1016	6.9
	Random Selection	90	39	65	12	20.2
TreeMap	Shape Abstraction	106	20	281	1016	7.2
	Symbolic Shape Abstraction	102	13	1309	1016	6.2
	Random Selection	106	39	78	17	25.5

the optimal basic block coverage, but as expected for longer sequence lengths than the other techniques. In two cases, it also got the optimal predicate coverage, but for the other two it got considerably less than optimal. Again it is interesting to note that the two it fared worse in are the two examples with more complex environments (`BinomialHeap` and `FibHeap`). This supports the belief that random selection suffers when the environment is not only large, but only a (small) subset of the options will produce the desired result: note that the environment for basic block coverage is the same as for predicate coverage, but predicate coverage requires very particular sequences to obtain high coverage.

It is probably human nature to want the simplest solution also to perform the best. With this in mind we also wanted to improve on the results obtained for random selection. It was conjectured that picking `M = N` might adversely affect random search since the number of choices increase at higher values of `M` and `N` although the space of useful values (i.e. the ones that can obtain the optimal coverage) increase much slower. Therefore we repeated the experiments for predicate coverage with `M` being fixed at the value we found the coverage from Table 4, but with `N` being brought down from `M` to the smallest value we knew could still get the coverage according to the other results obtained. For example for `TreeMap` this meant running with `M = 39` and `N` varying from 39 down to 20 (a conservative lower-bound seen for the shape abstraction analysis).

Note that these results are somewhat biased towards getting good results for random selection. Ironically, the results indicate that random selection is less effective, when we add one more dimension to the quality criteria, namely, frequency of highest coverage obtained. Originally we measured the coverage once during any run of the random selection, and that is what is reported in Table 4. However doing all the additional experiments, and taking into account only runs that *could* obtain the optimal coverage we found that the percentage chance of getting the optimal coverage for random (note, not necessarily the optimal coverage for all techniques) were as follows: 1.5%(6/400) for `TreeMap`, 0.38%(2/520) for `FibHeap` and 0.17%(1/600) for `BinomialHeap`. Note that again the reduction in chance of finding optimal coverage follows the order of the complexity of the environments, where `TreeMap`'s environment is simpler than `FibHeap`'s environment which in turn is simpler than `BinomialHeap`'s environment.

Considering the likelihood of obtaining the optimal results, random search performed poorly for obtaining high predicate coverage. We believe that the reason is that the input search space is complex/large and only a selected subset gives optimal results; in these situations the techniques based on shape abstractions yielded superior results.

5.3.5 Complex data structures as input parameters

Here we did not consider API calls that take structured data as input. In prior work [30] we analyzed `TreeMap` with structured inputs with a black-box test input generation technique similar to Korat [7] and a white-box symbolic execution technique based on lazy initialization. These techniques would be applicable in the current evaluation too. However, we believe that it is unlikely that they would scale

to large enough structures to get the optimal coverage obtained here. Furthermore, we believe that random search will also not work well for complex data, since the domain of possible structures will be very large, whereas the subset of these that are valid structures will be small. However, we need to do more experiments to validate these claims.

5.3.6 *Challenge*

In addition to the techniques and results reported here we also performed some experiments using the heuristic search facilities in JPF [13]. For the most part, the results were similar to what was reported here. However, using heuristic search for `FibHeap` with symbolic shape abstraction produced predicate coverage of 141 for sequence length 23 (better coverage than any other result here). This result indicates that there are more interesting test input generation techniques that need to be explored. In light of this we will make all our sources available on the JPF open-source website [19] for others to try additional techniques.

6. RELATED WORK

The work related to the topic of this paper is vast, and for brevity we only highlight here some of the closely related work. The most closely related works to ours are tools (and techniques) that generate test sequences for object oriented programs. We summarize them first.

JTest [20] is a commercial tool that generates test sequences for Java classes using "dynamic" symbolic execution, which combines concrete and symbolic execution over randomly generated paths. Unlike our work, this tool generates tests that may be redundant (exercise the same code), with little guarantees in terms of testing coverage. However, as we have seen in our experiments, random selection turns out to be pretty effective.

The AsmLT model-based testing tool [12] uses concrete state space exploration techniques and abstraction mappings, in a way similar to what we present here. Rostra [32] also generates unit tests for Java classes, using bounded-exhaustive exploration of sequences with concrete arguments and abstraction mappings. While both these tools require the user to provide the abstraction mappings, we provide automated support for shape abstractions that we have found very useful (see the experiments).

In previous work [30], we showed how to use model checking and symbolic execution to generate test inputs to achieve structural coverage for code that manipulates complex data structures, such as `TreeMap`. The approach was used in a black-box fashion (but it required an input specification written as a Java predicate – similarly to the Korat [7] tool) or in a white-box fashion (in which case only the source code for the method under test was needed). However, this approach was not used to generate method sequences (as we do here), but rather to build complex input structures that exercise the analyzed code in the desired fashion. Similarly, [4] discusses techniques to build complex input for testing red black tree implementation. On the other hand, Symstra [33] is a test generation tool that uses symbolic execution and state matching for generating test sequences for Java code – this is similar to our technique that uses symbolic execution and subsumption checking. Our paper contributes a novel combination of symbolic execution with abstraction, evaluation on Java container classes in terms of predicate coverage and comparison with random testing.

In a short paper [29], we discuss the use of explicit state model checking and abstractions for generating input test sequences for red black trees. In this paper we extend that work in several ways: we discuss the use of abstraction in the context of symbolic execution for test input generation and we provide an extensive evaluation using several Java container implementations (in addition to red black trees).

The Korat [7] tool, see also TestEra [23], supports nonisomorphic generation of complex input structures. Unlike the work presented here, this tool requires the availability of constraints representing these inputs. Korat uses constraints given as Java predicates (e.g. `repOK` methods encoding class invariants). Similarly, TestEra [23] uses constraints given in Alloy to generate complex structures.

The ASTOOT tool [9] requires algebraic specifications to generate tests (including oracles) for object oriented programs. The tool generates sequences of interface events and checks whether the resulting objects are observationally equivalent (as specified by the specification). Although here we were only interested in generating test sequences, using an algebraic specification to check the functional requirements of the code is a straightforward extension.

A set of techniques, not investigated in this paper, use optimization based techniques (e.g. genetic algorithms) for automated test case generation [27, 5]. In the future, we plan to compare these optimization based techniques with the state matching based techniques that are implemented in our framework.

The work presented here is related to the use of model checking for test input generation [1, 10, 15, 17]. In these works, one specifies as a (temporal) property that a specific coverage cannot be achieved and a model checker is used to produce counterexample traces, if they exists, that then can be transformed into test inputs to achieve the stated coverage. Our work shows how to enable an off-the-shelf model checker to generate test sequences for complex data structures. Note that our techniques can be implemented in a straightforward fashion in other software model checkers (e.g. [11, 8]).

Recently two popular software model checkers, BLAST and SLAM, have been used for generating test inputs with the goal of covering a specific predicate or a combination of predicates [6, 3]. Both these tools use *over-approximation* based predicate abstraction and use some form of symbolic evaluation for the analysis of (spurious) abstract counterexamples and refinement. We use *under-approximation* based abstraction – hence no spurious behavior is explored. The work in [3] describes predicate coverage as a new testing metric – we use a simplified version here; [3] also describes ways to measure when the optimal predicate coverage has been achieved (this a direction for future work that we would like to pursue).

The idea of using abstractions during model checking has been explored before. In [16], the abstractions need to be provided manually, while [24] uses automatic predicate abstraction for state matching during the the explicit execution of concurrent systems. In [2] we present a method for checking subsumption during symbolic execution; furthermore, shape abstractions are used to prune the search state space. Subsumption checking in [2] is more general than here (it handles partially initialized heap structures and summary nodes). However the abstractions in [2] can only handle lists and arrays, and not tree structures as we do here.

There is a lot of work on comparing random with partition testing in terms of cost effectiveness, e.g. [31, 14]. The verdict is still uncertain: [31] seems to suggest that random testing could be superior to partition testing under certain assumptions, while [14] suggests that, under different assumptions, partition testing is superior. Although fairly preliminary, we hope that our experiments will shed some light on this controversy, in the context of testing of data structures. Abstract matching can be seen as a form of partition testing (the state space explored by the model checker is partitioned according to the abstraction mapping) and seems superior to the other techniques.

7. CONCLUSIONS

We presented test input generation techniques that use state matching to avoid generation of redundant tests. The techniques range from exhaustive techniques such as classic model checking and symbolic execution with subsumption checking, through lossy abstraction techniques that use the shape of a container for state matching. We evaluated the techniques in terms of testing coverage achieved by the generated tests and we also compared them to random selection.

For the simple basic block coverage the exhaustive techniques are comparable to the lossy ones while for predicate coverage (which is more difficult to achieve) the lossy techniques fared better at obtaining high coverage. Random selection performed well except on the examples that contained complex input spaces, where the shape abstractions performed better. However, one should not lose sight of the strong guarantees that an exhaustive search, such as symbolic execution with subsumption, can provide: up to the maximum sequence length that allows exhaustive analysis, one can show that the implementation is free of errors.

For the future, we plan to investigate whether the shape abstraction that proved to be effective here, will also work for generating tests for other (more general) Java programs. We also plan to investigate other abstractions for our framework, e.g. abstractions used in shape analysis [26], and we plan to extend our evaluation to Java methods that take complex data structures and arrays as inputs.

Another direction for future research is to investigate the use of predicate abstraction for the automatic generation of different abstraction mappings. Towards this end, we plan to extend our work on automatic derivation of under approximation based abstractions [24], where we used a (backward) weakest precondition based calculation for automatic abstraction refinement. In the current setting we plan to adapt this algorithm to use (forward) symbolic execution.

8. REFERENCES

[1] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE International Conference on Formal Engineering Methods*, 1998.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. 13th International SPIN Workshop*, 2006.

[3] T. Ball. A theory of predicate-complete test coverage and generation, 2004. Microsoft Research Technical Report MSR-TR-2004-28.

[4] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *Softw. Test., Verif. Reliab.*, 10(3):149–170, 2000.

[5] A. Baresel, M. Harman, D. Binkley, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering (ICSE)*, 2004.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[8] 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.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, 1994.

[10] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999.

[11] 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.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 112–122, July 2002.

[13] A. Groce and W. Visser. Heuristics for model checking Java programs. *STTT Journal*, 6(4), December 2004.

[14] W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, 1999.

[15] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, D. George, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, Oct. 2003.

[16] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. 11th International SPIN Workshop*, 2004.

[17] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Grenoble, France, April 2002.

[18] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, Nov. 2001.

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

[20] JTest. `http://www.parasoft.com/jsp/home.jsp`.

[21] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] 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.

[24] C. S. Păsăreanu, R. Pelanek, and W. Visser. Concrete model checking with abstract matching. In *Proc. 17th International Conference on Computer Aided Verification (CAV)*, 2005.

[25] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), Aug. 1992.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[27] P. Tonella. Evolutionary testing of classes. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2004.

[28] 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.

[29] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red black trees using abstraction (short presentation). In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[30] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation in Java Pathfinder. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[31] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[32] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE)*, 2004.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005.