Test Input Generation for Red-Black Trees using Abstraction

Willem Visser and Corina S. Păsăreanu
NASA Ames Research Center
Moffett Field, CA 94035, USA
{wvisser,pcorina}@email.arc.nasa.gov

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

ABSTRACT

We consider the problem of test input generation for code that manipulates complex data structures. Test inputs are sequences of method calls from the data structure interface. We describe test input generation techniques that rely on state matching to avoid generation of redundant tests. *Exhaustive techniques* use explicit state model checking 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 a Java implementation of red-black trees.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Testing and Debugging—Testing Tools

General Terms: Algorithms, Verification

Keywords: Testing Object-oriented Programs, Model Checking, Abstraction, Coverage, Red-Black Trees

1. INTRODUCTION

Almost all large software systems contain portions of code that manipulate complex data. This is all the more true today since object oriented programming languages are becoming more and more popular. It is imperative for the reliability of these systems that this code is tested adequately. The most time consuming aspect of unit testing for this kind of software is the generation of sequences of API calls that cover the relevant structural and behavioral aspects of the code. This process is difficult due to the fact that the software under test (SUT) has "state" and it will react differently depending on this state when a new call is made.

In this paper, we address the problem of automated generation of such sequences of API calls to ensure high degrees of code coverage. Many approaches for doing such automated test sequence generation have been proposed - see Section 5. Here we consider one class of search algorithms that seems particularly promising for generating test sequences to achieve high code coverage: algorithms that use state matching to avoid generation of redundant tests. Essentially the idea is to exhaustively try all combinations of API calls and parameters to these calls up to a specified limit, but after each call the state of the SUT is analyzed to see if the "same" state has been seen before; if so, that sequence is discarded, if not the search continues with the next call. During this search the code coverage is measured and whenever new coverage is obtained the sequence of calls to achieve that coverage is recorded.

Our contribution is an integrated framework that uses state matching for *automated* test generation. The framework also incorporates a technique based on random selection - to be used as a point of comparison with state matching techniques. We show that there are a number of different options for the state matching and not all of them need to be a precise/complete matching, i.e. two states that are not exactly the same can be considered equivalent and that can improve the efficiency of the search - although it is a *lossy* search since parts of the feasible input space can be discarded. In particular we show that a matching on the structure or shape of a container is a very efficient way to achieve good coverage. We use the Java PathFinder [11, 13] model checker as the basis for building the test generation framework. We evaluate the framework on a Java implementation of red-black trees.

```
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

2. EXAMPLE

We illustrate our test generation framework on a Java implementation of a binary search tree (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.

```
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 2: Environment for concrete search

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`:

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

Typically, checking the correctness of executions for such test inputs relies on design-by-contract annotations translated into run-time assertions. One can also check class invariants (`repOK` predicates [3]) or just absence of run-time errors (e.g. absence of uncaught exceptions).

Our approach integrates several techniques for test input generation in a unified model checking framework. The approach requires the user to produce an environment, i.e. a test driver for the Java implementation (the system under test `SUT`). In this paper, we consider nondeterministic environments that execute all sequences of API method calls up to a user-specified size `M`. The user also needs to specify the range of values for the method parameters `[0, N-1]`.

The model checker analyzes the composition of the container and the environment and it generates sequences that achieve the desired testing coverage. We use basic block coverage, as a representative example of a widely used structural coverage measure. We also consider a simple form of predicate coverage [2] that measures whether all combinations of a predetermined set of predicates are covered at each basic block. The user may choose between several techniques that our framework implements (they are described in detail in the next section).

3. TEST GENERATION TECHNIQUES

"Classical" Exhaustive State Space Search We illustrate this technique using the `BinTree` example introduced in the previous section. The testing environment is illustrated in Figure 2. The environment contains special JPF annotations (`Verify`): `beginAtomic()` ...`endAtomic()` specify that the execution of the enclosed block should proceed atomically; `random(N-1)` returns values $[0, N-1]$ nondeterministically.

By default, JPF stores all the explored states (and it backtracks when it visits a previously explored state). This

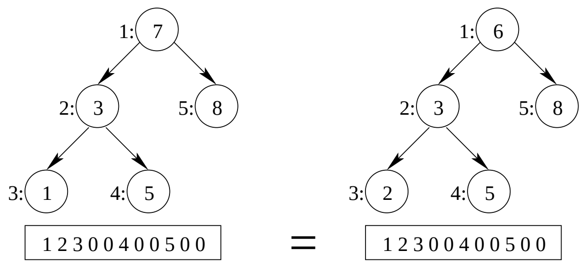

Figure 3: Abstraction recording shapes

straight-forward 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 concrete execution with abstract matching as described below.

Concrete Search with Abstract Matching The idea is to use the model checker to perform the concrete execution of all the possible method sequences (as above) but to store *abstract* versions of the concrete 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 2, which includes statement 8: `Verify.ignoreIf(store(abstractMap(t)))`.

`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. The user may choose from several default abstractions that are provided by JPF or may create new abstraction mappings.

One default abstraction that we have found useful records only the (concrete) heap shape of a container, while it abstracts away the data fields from each container element. This abstraction is illustrated in Figure 3, 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 3), and are obtained through a process called *linearization* [10, 15]. 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;

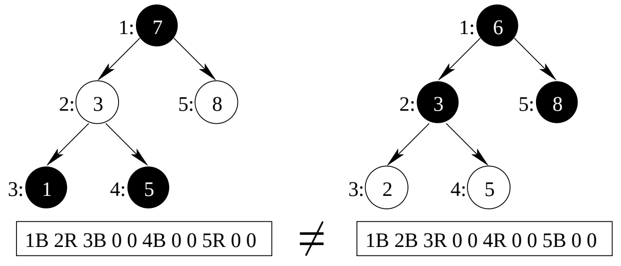

Figure 4: Abstraction recording shapes and colors

null pointers have values 0. Comparing shapes reduces to comparing sequences.

This abstraction can be made more precise by extending it to record information about the container elements. For example, consider two *red-black trees* in Figure 4. Red-black trees are binary search trees with one extra bit of information per node: its *color*, which can be black or red (in Figure 4, filled circles denote black nodes while empty circles denote red nodes). In Section 4, we will analyze red-black trees as given in the `java.util.TreeMap` library.

Figure 4 also shows the new abstract representations, which augment the shape with the color values. Note that the two trees will no longer be considered to be the same (although they have the same shape). The abstraction for red-black trees can be augmented further, to also encode the actual concrete `elem` values of each node. Here is an example: 1B7 2R3 3B1 0 0 4B5 0 0 5R8 0 0.

In general, the user can specify which data fields to be added to the linearization of an object. When all the fields are selected the state matching is complete and it can form the basis of an exhaustive technique (see Section 4); the approach is similar to the linearization used for representing the complete concrete heap (shape plus data), to achieve heap symmetry reduction in model checking [10].

Random Search The environment that we use for random search is similar to the one presented in Figure 2, 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 standalone 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. EVALUATION

As mentioned, we used the JPF model checking tool to implement our testing framework. We used the *listener* mechanism [11] 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. For the abstraction mappings the user can select the fields to be added to the linearized nodes of a structure.

As a system under test we used a Java implementation of red-black trees (from `java.util.TreeMap`). The Java methods were instrumented to measure basic block coverage (which implies statement coverage). At each basic block,

Basic Block Coverage: Exhaustive Techniques

	Cov.	Seq.len.	Time	Mem	Avg.len.
Model Checking	37	6	38s	243M	4.2
Complete Abs.	39	7	9s	34M	4.3

Basic Block Coverage: Lossy Techniques

	Cov.	Seq.len.	Time	Mem	Avg.len.
Shape Abs.	39	10	2s	6M	4.6
Random Search	39	10	18s	5M	7.1

Predicate Coverage: Exhaustive Techniques

	Cov.	Seq.len.	Time	Mem	Avg.len.
Model Checking	55	6	38s	229M	4.5
Complete Abs.	95	10	271s	844M	5.8

Predicate Coverage: Lossy Techniques

	Cov.	Seq.len.	Time	Mem	Avg.len.
Shape Abs.	106	22	182s	98M	6.9
Random Search	106	39	78s	17M	25.5

Table 1: Results for exhaustive vs. lossy test generation techniques for `TreeMap`.

we also measure the coverage of all the combinations of a set of predicates chosen from conditions in the source code. The container class is augmented with an environment as described in Section 3.

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 and concrete search with complete abstract matching (i.e. linearization of a structure with all fields included). Lossy techniques include: concrete search with abstract matching based only on shape and random search. The results are summarized in Table 1 (JPF is used with breadth first search order). We report coverage (Cov.), test sequence length (Seq.len.), time (seconds), memory (MB), and average test sequence length (Avg.len.). 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. 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.

We have experimented with different abstraction mappings, for example an abstraction recording the shape and color for `TreeMap` (see section 3); we only report here the results for two abstractions, where we use either just the shape or the shape with all the fields. Note that using all the fields in a structure gives an exhaustive search, since the contents of the container uniquely identifies its state - for example, the length of the sequence of calls has no bearing on the behavior of the container.

Exhaustive vs. lossy search The results indicate that all the lossy techniques achieved the optimal basic block coverage (39) and where comparable, they achieved it faster and with less memory than the exhaustive techniques.

Abstract matching State matching based on the shape abstraction achieves the highest coverage, for the shortest sequences. Due to abstraction, it consumes less memory than other techniques. Only random search, that essentially has no memory footprint, uses less memory when coverage is the same (but for longer test sequences). The complete abstraction that takes the shape and all fields into account performs well, but uses more time and memory. Note that this technique performs consistently better than "classic" model checking which is closely related. We conjecture that for the analyzed `TreeMap` implementation, the shape is an accurate representation of the container state and hence the shape abstraction is appropriate here. It remains to be seen whether this will hold for general programs.

Random search Random search achieved both optimal basic block coverage and optimal predicate coverage, but as expected for longer sequence lengths than the other techniques. However, we believe that random search will begin to suffer once we consider more complex environments (and methods with complex input parameters).

5. 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 AsmLT model-based testing tool [7] uses concrete state space exploration techniques and abstraction mappings, in a way similar to what we present here. Rostra [15] 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 several shape abstractions (see the experiments).

The Korat [3] tool, supports non-isomorphic 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).

The work presented here is related to the use of model checking for test input generation [1, 5, 8, 9]. Model checking lends itself well to test input generation, since one can specify as a (temporal) property that a specific coverage cannot be achieved and the model checker will report a counterexample trace, if it exists, that then can be transformed into a test input 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. [6, 4]).

6. CONCLUSIONS

We described and compared a number of test input generation techniques all based on state matching. We measured the techniques in terms of coverage achieved by the generated tests. Although for the simple basic block coverage the exhaustive techniques are comparable to the lossy ones, for predicate coverage (which is more difficult to achieve), the lossy techniques are better at obtaining high coverage. However, one should not lose sight of the strong guarantees that an efficient exhaustive search can provide: up to the maximum sequence length that allows exhaustive analysis, one can show that the implementation is free of errors.

The techniques presented here only considered concrete data, but it has been shown that using symbolic data allows efficient test input generation for code manipulating complex data [12, 14, 16]. State matching during symbolic execution however requires subsumption checking. We plan to extend the current framework to also address symbolic execution with efficient subsumption checking.

We only focussed 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.

7. REFERENCES

[1] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the 2nd IEEE ICFEM*, 1998.

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

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

[4] 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 ICSE*, June 2000.

[5] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. of the 7th ESEC/FSE.* Springer-Verlag, 1999.

[6] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual POPL*, Paris, France, Jan. 1997.

[7] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. of ISSTA*, July 2002.

[8] 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. of 3rd FATES*, Montreal, Canada, Oct. 2003.

[9] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th TACAS*, Grenoble, France, April 2002.

[10] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. 16th ASE*, Nov. 2001.

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

[12] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, 2003.

[13] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of 15th ASE*, Grenoble, France, 2000.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation in java pathfinder. In *Proc. of ISSTA*, 2004.

[15] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. of 19th ASE*, 2004.

[16] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Proc. of *TACAS*, 2005.