On the Effect of Object Redundancy Elimination in Randomly Testing Collection Classes

Pablo Ponzio
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Valeria Bengolea
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Simón Gutiérrez Brida
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Gastón Scilingo
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Nazareno Aguirre
Dept. of Computer Science,
University of Rio Cuarto, Argentina

Marcelo Frias
Dept. of Software Engineering,
Buenos Aires Institute of Technology,
Argentina

ABSTRACT
In this paper, we analyze the effect of reducing object redundancy in random testing, by comparing the Randoop random testing tool with a version of the tool that disregards tests that only produce objects that have been previously generated by other tests. As a side effect, this variant also identifies methods in the software under test that never participate in state changes, and uses these more heavily when building assertions.

Our evaluation of this strategy concentrates on collection classes, since in this context of object-oriented implementations that describe stateful objects obeying complex invariants, object variability is highly relevant. Our experimental comparison takes the main data structures in java.util, and shows that our object redundancy reduction strategy has an important impact in testing collections, measured in terms of code coverage and mutation killing.

1 INTRODUCTION
Software testing is widely recognized as an important mechanism for software quality assurance [3, 8, 10], and due to the inherent difficulty and cost of its systematic application, techniques for automated test generation have received significant attention [2, 4–6, 11, 12, 15, 16]. Evaluating automated test generation techniques for automatic test generation have received significant attention [2, 4–6, 11, 12, 15, 16]. Evaluating automated test generation techniques is also challenging, and often demands selecting appropriate case studies for test generation techniques assessment, especially for test generation techniques that are somehow affected by scalability issues. In this context, the implementation of collection classes, such as lists, sets and maps, has been extensively used [2, 4, 16].

Collection implementations are interesting for testing for various reasons. Firstly, they can be tested in isolation of large system libraries and other dependencies. Secondly, they are also relatively small software components, whose complexity is in the structure of the code, the conditions involved, and the invariant properties of the objects they describe. These reasons make them suitable for techniques that focus on precisely this kind of complexity, such as symbolic execution based approaches [16], or approaches that exploit class invariant specifications [4, 7]. Also, object-oriented implementations of collections, such as those that are typically the subject of automated testing evaluations, are paradigmatic examples of object oriented programming, that feature clear and clean component interfaces that hide intricate implementation details, describe stateful objects that (should) obey complex invariants, but that are also (at least theoretically) unbounded, leading to very large or even infinite testing domains.

While collection classes have been the subject of so-called “systematic” automated testing techniques, it has also been shown that random testing can be rather effective in testing collections, with performances that are comparable, for many data structures, with the most effective systematic techniques [14]. In this paper, we evaluate a variant of feedback-directed random testing, as realized by the Randoop tool [12], for testing collection classes. This variant incorporates a mechanism to check test redundancy, based on the idea of considering a test redundant if it only produces objects that have already been produced in previous tests. By using this mechanism one diminishes object redundancy in tests, preventing one to save tests that produce objects that have been produced before. Moreover, as we will explain, this mechanism helps in distinguishing methods that produce object updates from those that do not, which allows us to provide a specialized treatment for observer methods, using them more extensively in generating assertions. We compare this technique with standard feedback-directed random testing, and show that, in the context of container classes, object redundancy elimination has a significant impact in the quality of the generated suites, achieving an important margin in code coverage and mutation killing, over the main classes in java.util.

2 RANDOM TESTING WITH OBJECT REDUNDANCY CONTROL
Random testing is the process of evaluating software on randomly produced tests [5, 11, 12]. Randomly testing software whose inputs
are numeric, or in general from basic datatypes, is straightforward; but doing so on more complex types, in particular class-based objects, calls for more sophisticated mechanisms. Various approaches to randomly testing software with complex inputs have arisen, including some based on defining generators (e.g., [5]) and some that exploit the classes’ programming interface for object generation [11, 12]. Among the latter, feedback-directed random testing, as implemented in the Randoop tool, has been particularly successful. The generation mechanism works as follows. For every datatype, a set of sequences that produce inputs of such datatype is maintained. To start with, a set of initial values is considered (e.g., some initial common values for basic datatypes, null for reference types, etc.). Then, to build a new test sequence one starts by randomly selecting a method \( m \), among all methods in the software under test, and randomly choosing, for each of the parameters of \( m \), test sequences of the corresponding types, from the already collected ones. The new test sequence is simply the sequential composition of the sequences for all parameters, with the call to \( m \) with the generated parameters as a last statement. As an example, consider class BinarySearchTree, a heap-allocated implementation of dictionaries (sets) over binary search trees with the following methods:

- public BinarySearchTree(), a constructor that builds an empty dictionary.
- public boolean insert(int elem), an insertion routine that adds a new element to a dictionary, returning true iff the element did not already belong to the set.
- public boolean remove(int elem), removes an element from a dictionary, provided it belonged to the set (it returns true if removal succeeded, false if the element did not belong to the set).
- public boolean search (int elem), that searches for an element in the dictionary, and returns true iff it finds it.
- public int smallest(), retrieves the smallest element in the set, provided it is not empty.
- int size(), returns the number of elements in the set.
- boolean isEmpty(), returns true iff the dictionary has no elements.
- void removeAll(), removes all the elements in the set.

Assume that the random test generation process starts with initial values \{0, 1, 100\} for integers, \{true, false\} for booleans, and null for BinarySearchTree. Suppose also that the first randomly selected method for generating a new test sequence is the constructor, BinarySearchTree(). Since this method has no parameters, the process does not need to provide values for parameters, and a new test sequence, generating a BinarySearchTree object, is built containing only this method call. Now suppose that the randomly selected method in the second iteration of the generation process is insert(int elem); both a BinarySearchTree object (the receiving object) and an integer value (argument elem) are required to build a test. Assuming that the randomly selected values/sequences for these arguments are the constructor and value 100, respectively, the new test sequence is the following:

```java
BinarySearchTree t0 = new BinarySearchTree();
int elem0 = 100;
boolean b0 = t0.insert(elem0);
int elem1 = t0.smallest();
```

This sequence (let us call it test1) produces a BinarySearchTree object, the indirect output of insert, as well as a boolean value (true). The sequence is saved for future test generation iterations. Randoop’s feedback-directed generation consists of running a test sequence as soon as it is produced [12]. In this way, the tool can check whether the test fails or not. If it does not fail, the return value is saved and an assertion checking that the obtained value is the result of the call, is added for regression. Failing tests are discarded for generation: no valid test can be produced by extending a failing test. For instance, if the above sequence is used for the generation of a new test sequence (let us call this new one test2), as the following:

```java
BinarySearchTree t0 = new BinarySearchTree();
int elem0 = 100;
boolean b0 = t0.insert(elem0);
int elem1 = t0.smallest();
```

and the following assertions are added to the test:

```java
assertTrue(b0 == true);
assertTrue(elem1 == 100);
```

Of course, some produced sequences may fail, e.g., if one invokes smallest on an empty dictionary. In such cases, the failing behaviour is captured as a test that is saved (let us call it test3), but the produced sequence is not added to the set of sequences for continuing with test generation (failing tests should not be used as part of newly created tests).

To motivate our approach to reduce object redundancy, let us mention the following. When using Randoop to randomly generate tests for BinarySearchTree for 3 seconds, 1307 tests are generated, summing up 61801 lines of code. These tests produce 10144 objects, but only 106 different objects. That is, there is a high degree of redundancy in the number of objects involved in the tests.

Let us now present our approach to reduce object redundancy. Consider the following additional test for the dictionary implementation:

```java
BinarySearchTree t0 = new BinarySearchTree();
int elem0 = 100;
boolean b0 = t0.insert(elem0);
int elem1 = t0.smallest();
int elem2 = t0.smallest();
```

This test (let us call it test4) shares a significant part with test2. In fact, since smallest is an observer, this test is not exercising any new behaviour with respect to test2. Our approach to avoid generating test4 will be simple: tests of the kind of test2, that do not produce new objects with respect to previous tests (notice that this test does not generate anything new compared to test1) are kept but not extended. In this sense, they receive a similar treatment to failing tests in standard Randoop. Similarly, tests such as:

```java
BinarySearchTree t0 = new BinarySearchTree();
int elem0 = 100;
boolean b0 = t0.insert(elem0);
t0.removeAll();
```

(let us call it test5) are produced and stored, but not used for generating new tests, since, again, it does not produce any new objects.

There is a difference between tests test4 and test5, however: the former does not produce any state change in the last statement, while the latter does change the state. Notice then that while tests
are produced and executed, one can very straightforwardly classify methods: initially all methods are unclassified, and as soon as a method is found to produce a state change, it is classified as a modifier. After generating tests for some time, all methods that remain unclassified are deemed observers (they have not participated in any state change in the generated tests), and can be used to complement generated tests with further assertions, using these “observers”. This is our generation process, that as it can be seen, consists of two parts: generation (and method classification), and tests extension with further assertions.

Notice that our generation approach requires two mechanisms that standard Randoop did not need: one for checking whether a test produces an object that has not been produced before, and one for checking whether the last statement of a test produces a state change. For the latter, we explicitly observe the states before and after the last statement in a test. But for the former, since a precise implementation of object redundancy checking requires storing all produced objects, and this quickly becomes infeasible, we consider instead a significantly cheaper approach, that deems a test redundant if it does not involve any new values for object fields. More precisely, as tests are generated, the field extensions [13] (values that fields received in the generated objects) of all fields in the software under test are increasingly built; a test is considered redundant if it does not contribute any new value to any field, i.e., if it did not extend the current field extensions.

3 EVALUATION
Let us evaluate the described technique on the main data structures in java.util, more precisely: Linked List (LList), a doubly linked list implementation of lists, an Array List implementation (AList), maps on hash tables (HMap), and maps on red-black trees (TMap). The classes evaluated are exactly those in java.util, in Java JDK 1.7, without any alterations. We ran both standard Randoop (Rand) and Randoop with object redundancy elimination (R. Elim) on these case studies, with various increasing limits in test suite sizes (Limit). Shown results correspond to the average of three runs for each tool. All experiments were run on 3.2GHz quad-core Intel Core i5-4460 machines, with 4GB of RAM. We measure and report generation time, test suite size (differs from the corresponding limits due to test subsumption performed by Randoop), number of different objects generated by the suites, and test suite quality in terms of statement coverage, branch coverage and mutants killed.

Let us summarize the results. Firstly, it is important to notice that our technique’s generation times are comparable with standard Randoop, despite the fact that our approach involves some important overhead. We conjecture this has to do with the size of the evaluated classes and the limit for suite size, which are both relatively small. We have observed an overhead of roughly 2X compared with standard Randoop, when larger case studies (in particular, the defects4j projects) are evaluated. Secondly, for essentially the same suite size, the degree of redundancy is greatly reduced by our technique, which is not surprising, since it was the aim of the approach. Finally, the quality of the produced suites is in general improved by redundancy elimination (with the exception of Linked List, where statement/branch coverage and mutants killed saturates at around limit 10000). In some cases, in particular HashMap and TreeMap (the most complex data structures analyzed), the improvement in coverage and mutants killed is rather notorious.

4 RELATED WORK
The problem of producing redundant test cases is an important problem in automated test generation, and random testing approaches, as well as other techniques, attempt to tackle it. Randoop in particular exploits feedback, avoiding the extension of failing tests, and finally performing a subsumption analysis to reduce test suites, discarding tests that are part of other, larger tests [1]. Both object redundancy elimination and method classification are not, as far as we are aware of, part of any random testing technique.

Other test generation techniques, such as those driven by white-box criteria, such as Symbolic PathFinder [16], Pex [15] and UDITA [9], also try to reduce test redundancy. They do so by incorporating techniques that force them to produce a single test per criterion’s equivalence class. Thus, they tend to produce suites where different tests exercise different branches, statements, bounded paths, etc. These mechanisms are tightly coupled to coverage-driven testing, and are difficult to transfer to random testing.

5 CONCLUSION
We have assessed a technique to improve feedback-directed random test generation, that incorporates a mechanism for reducing test suite redundancy. This mechanism is based on discarding tests that only produce objects that have already been observed in previous tests. The technique also provides an on-the-fly classification of methods as modifiers and observers, and exploits the latter for more heavily building assertions. Our evaluation, based on a benchmark of collection classes, shows that in this context this technique produces test suites with less redundancy in terms of the objects they involve. Moreover, at the same time the generated suites achieve important improvements in quality, compared to standard feedback-directed random testing, measured in terms of coverage, mutation killing and bug finding.

Random testing has been found to compete with more complex techniques, in testing collection classes. However, on the more complex collections, such as those based on red-black trees and similar structures, random testing is outperformed by some more systematic (e.g., coverage-driven) techniques. The improvement we evaluated in this paper may have the potential to boost random testing in this context, and we plan to investigate this further. We also plan to evaluate the impact of object redundancy elimination in other random testing approaches, as well as its consequences in other aspects of test suite quality, such as test readability.

REFERENCES
Table 1: Comparison between Randoop and our Technique, in terms of generation time, suite size, number of different objects produced, statement coverage, branch coverage, and mutants killed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>