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 obbeying 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.

ACM Reference Format:
Pablo Ponzio, Valeria Bengolea, Simón Gutiérrez Brida, Gastón Scilingo, Nazareno Aguirre, and Marcelo Frias. 2018. On the Effect of Object Redundancy Elimination in Randomly Testing Collection Classes. In *SBST'18: IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194718.3194724

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 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 in 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) obbey 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

SBST'18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194724

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 arised, 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:

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

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:

```
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:

```
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:

```
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 exercizing 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:

```
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 defects4f 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

[1] Web site of the Randoop test generation tool. https://randoop.github.io/randoop/.

[2] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[3] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

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.

	Limit	Gen. time		Suite Size		No. of Objects		Stmt. Cov.		Branch Cov.		Mut. Killed	
		Rand.	R. Elim.	Rand.	R. Elim.	Rand.	R. Elim.	Rand.	R. Elim.	Rand.	R. Elim.	Rand.	R. Elim.
	100	0	0	80	81	50	85	214	231	84	96	116	149
	200	0	1	166	168	77	135	248	249	102	106	143	164
LList	500	1	2	437	442	141	276	260	263	114	115	161	182
	1000	2	5	919	914	224	455	263	265	116	116	171	185
Stmt. 325	2000	5	7	1896	1882	365	749	265	266	118	118	181	188
Mut. 301	3000	8	13	2871	2861	492	967	266	266	118	118	185	189
Branches. 114	5000	13	18	4844	4838	721	1321	267	266	119	118	186	190
	10000	36	31	9858	9819	1232	1897	267	267	119	119	190	191
	100	0	0	89	88	50	63	139	145	53	62	143	144
	200	0	0	171	178	61	84	145	173	57	76	154	195
AList	500	1	2	455	455	90	163	166	184	72	86	193	229
	1000	2	4	926	924	124	295	178	188	80	89	210	266
Stmt. 380	2000	6	7	1887	1868	175	534	183	202	85	95	225	285
Mut. 718	3000	8	9	2883	2818	218	780	184	202	87	95	233	296
Branches. 156	5000	18	14	4768	4733	277	1204	184	203	87	95	242	301
	10000	31	30	9611	9564	428	2141	190	203	90	96	254	308
	100	0	0	87	85	80	85	133	174	54	86	73	100
	200	0	0	174	174	123	131	170	199	80	107	95	130
HMap	500	1	1	457	452	236	239	190	201	102	112	124	160
	1000	3	1	918	924	369	398	199	203	112	119	143	181
Stmt. 360	2000	6	2	1891	1877	611	701	203	208	115	122	161	201
Mut. 566	3000	9	3	2863	2841	830	971	203	210	115	123	169	206
Branches. 230	5000	17	5	4810	4783	1217	1479	206	212	117	124	175	211
	10000	42	13	9692	9645	2041	2599	211	213	121	124	182	217
	100	0	0	87	85	64	76	206	347	69	174	72	206
	200	0	1	175	178	106	122	242	386	92	200	101	252
TMap	500	1	2	461	460	210	245	323	478	145	266	165	342
	1000	1	5	953	946	362	433	332	489	156	264	189	340
Stmt. 872	2000	3	7	1917	1929	592	794	359	508	177	276	217	358
Mut. 942	3000	5	10	2899	2914	801	1131	364	518	183	281	227	368
Branches. 522	5000	8	19	4892	4884	1190	1820	392	532	206	293	255	389
	10000	22	42	9895	9850	2001	3457	434	533	233	295	292	391

[5] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000.

[6] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.

[7] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010.

[8] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[9] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 225–234. ACM, 2010.

[10] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[11] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings*, volume 4362 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007.

[12] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[13] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Field-exhaustive testing. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 908–919. ACM, 2016.

[14] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: Random or systematic? In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6603 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2011.

[15] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[16] Willem Visser, Corina S. Pasareanu, and Radek Pelánek. Test input generation for java containers using state matching. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 37–48. ACM, 2006.