Testing Container Classes: Random or Systematic?

Rohan Sharma[1], Milos Gligoric[1], Andrea Arcuri[2], Gordon Fraser[3], and Darko Marinov[1]

[1] University of Illinois, Urbana, IL-61801, USA
{sharma27, gliga, marinov}@illinois.edu
[2] Simula Research Laboratory, Lysaker-P.O. Box 134, Norway
arcuri@simula.no
[3] Saarland University, Saarbruecken-66123, Germany
fraser@cs.uni-saarland.de

Abstract. Container classes such as lists, sets, or maps are elementary data structures common to many programming languages. Since they are a part of standard libraries, they are important to test, which led to research on advanced testing techniques targeting such containers and research on comparing testing techniques using such containers. However, these techniques have not been thoroughly compared to simpler techniques such as random testing. We present the results of a larger case study in which we compare random testing with shape abstraction, a systematic technique that showed the best results in a previous study. Our experiments show that random testing is about as effective as shape abstraction for testing these containers, which raises the question whether containers are well suited as a benchmark for comparing advanced testing techniques.

1 Introduction

Automation of test generation is an important and still open issue, regularly leading to new techniques and refinements of existing techniques. The empirical evidence on research in this area often focuses on container classes [4–7, 10, 12, 14, 17, 19, 22, 23, 26, 27, 29–31, 33, 34]—containers are an important part of many standard libraries, and bugs in these containers could significantly affect applications, so directing testing efforts to these containers is worthwhile. Testing containers is not only important but also challenging to achieve with some advanced testing technique such as those based on symbolic execution [30].

Automating testing for containers is convenient because they usually do not interact with the environment and can be tested without construction of complex input data [21]. Any results achieved on containers for one language easily carry over to other languages, as the data structures are generic and implemented for many different languages. However, precisely these aspects of containers also mean that simple techniques such as random testing could be able to achieve good results. Unfortunately, the literature offers little evidence on how more advanced techniques compare to random testing. In fact, excluding comparisons with search algorithms, we are aware of only one study that compared random testing to systematic techniques, by Visser, Pasareanu, and Pelanek [30]; we will refer to this study as the *VPP* study.

The VPP study proposed several advanced techniques for test input generation for Java container classes and compared these techniques against one another and with random testing on four container classes. The comparison metrics were basic block coverage and a simplified version of *predicate coverage* [11] that measures how many combinations of program predicates are covered (which differs from the traditional condition or MCDC coverages [3]). The results showed that among the advanced techniques the best was *shape abstraction* (described in detail in Section 4.2). The results in the VPP study also showed that shape abstraction was the same as random testing for basic block coverage but even better than random testing for predicate coverage.

In this paper, we perform a larger set of experiments to compare random testing and shape abstraction, which remains the best technique in systematic test generation for containers. Our study substantially extends on VPP in several important aspects:

Number of containers: We use a total of *13 different container classes* in our evaluation, four from VPP and nine more that were used previously in various other studies [17, 25].

Types of containers: We consider containers implemented with both *pointer-based, linked structures* and *array-based structures*, whereas VPP (and several other studies) used only containers implemented with pointer-based structures.

Metrics: We use *mutation score* [3], in addition to predicate coverage, for comparison of techniques. To the best of our knowledge, this is the first study that relates predicate coverage and mutation scores. We also measure predicate coverage more thoroughly than the VPP study which considered only a few manually selected program points whereas we use a semi automated tool to consider all branches. To make it easier for other researchers to experiment with predicate coverage, we made our instrumented code publicly available at http://mir.cs.illinois.edu/coverage.

Statistical Analysis: We perform a rigorous *statistical analysis* of the results, as opposed to VPP which had no statistical analysis.

Results: The experiments show that although random testing is much faster than shape abstraction, *random testing still achieved comparable predicate coverage and mutation scores to those achieved by shape abstraction*. Specifically, random testing was better for four containers, shape abstraction was better for five containers, and the results were inconclusive for four containers. In contrast, the VPP study found shape abstraction better than or equal to random testing for all four containers considered. Our experiments also raise the concern that containers should not be used as a *de facto* benchmark for comparing advanced testing techniques because random testing can work very well for containers.

Bugs: While the goal of our study was to compare random testing and shape abstraction but not necessarily look for bugs, we still *found three real bugs* in two containers used in previous studies [17, 25]. All three bugs were found by random testing, were missed by the advanced techniques used in previous studies, and were confirmed by the original authors of the respective container code.

One relevant aspect in which our study evaluates less than the VPP study is that we test only two basic methods/operations for each container (namely, *add* and *remove*), whereas the VPP study tested a larger number of methods/operations for two of their four containers.

Table 1. Sample recent papers that use containers among subjects in the case studies.

Authors	Year	Reference	#Subjects	#Containers	%Containers
Tonella	2004	[27]	6	5	83%
Visser *et al.*	2004	[29]	1	1	100%
Xie *et al.*	2004	[33]	11	9	81%
Xie *et al.*	2005	[34]	7	7	100%
Visser *et al.*	2006	[30]	4	4	100%
Wappler and Wegener	2006	[31]	4	4	100%
d'Amorim *et al.*	2006	[14]	16	12	75%
Inkumsah and Xie	2008	[19]	13	10	77%
Arcuri and Yao	2008	[10]	7	7	100%
Andrews *et al.*	2008	[4]	2	1	50%
Arcuri	2009	[6]	1	1	100%
Ribeiro *et al.*	2009	[22]	2	2	100%
Ribeiro *et al.*	2010	[23]	2	2	100%
Baresi *et al.*	2010	[12]	15	9	60%
Arcuri	2010	[7]	6	6	100%
Andrews *et al.*	2010	[5]	34	34	100%
Staats and Pasareanu	2010	[26]	6	4	66%
Galeotti *et al.*	2010	[17]	6	6	100%

2 Related Work

A number of studies compared advanced techniques for test generation or test selection with random testing [14–16,18,32], but these studies did not provide conclusive answers either way (sometimes random testing looked better and sometimes worse than more advanced techniques), and they did not focus on containers. Several recent techniques use random generation for object-oriented unit tests [1, 13, 21] but target shallower exploration of larger codebases and can generate complex test data inputs, while testing containers focuses on deeper exploration of smaller codebases and typically requires only simple data inputs. In this paper we focus on test generation for containers.

While clearly not all testing studies use only containers for evaluation [21], containers are still widely used in many recent studies on testing. Table 1 shows a sample of 18 papers that either propose techniques specifically for testing containers or use containers as subject code to evaluate new or existing testing techniques. As can be seen, containers are a large percentage of subjects used for evaluations, even when the number of subjects is not very high. Our evaluation uses 13 containers. While several of these studies evaluate effectiveness of random testing in various scenarios [5–7,10,14,30,33], only the VPP study [30] directly compares random testing and advanced systematic techniques (not based on search). In terms of metrics used, branch coverage is the most represented. The only exceptions are predicate coverage used in the VPP study [30], statement coverage [5], MCDC [26], and an unspecified "structural coverage" [22, 23]. We use not only predicate coverage, which subsumes branch coverage, but also mutation score. Only a few of these studies use statistical analyses [5–7, 10]. We also present a statistical analysis of our experimental results.

```
public class TreeSet {
    int size;
    TreeSetEntry root;

    public TreeSet() { ... }
    public boolean add(int aKey) { ... }
    public boolean remove(int aKey) { ... }
    ...
}

class TreeSetEntry {
    int key;
    boolean color;
    TreeSetEntry left;
    TreeSetEntry right;
    TreeSetEntry parent;
}
```

(a) Parts of the TreeSet class

```
public class TreeSetTest {
    public void test1() { // length 1
        TreeSet s = new TreeSet();
        s.add(5);
    }
    public void test2() { // length 1
        TreeSet s = new TreeSet();
        s.remove(5);
    }
    public void test3() { // length 2
        TreeSet s = new TreeSet();
        s.add(5);
        s.remove(21);
    }
    ...
}
```

(b) Example tests for TreeSet

Fig. 1. This is a typical example of a container class, where testing focuses on a few selected methods (*add* and *remove* in this case). A test case starts with the default constructor for the container, which creates an empty instance. Then, on this container the *add* and *remove* methods are repeatedly called. The length of the test case is the number of such calls.

3 Example

We next describe an example that illustrates the problem of test generation for containers. Figure 1(a) shows partial code for the `TreeSet` container class that we obtained from a previous study by Galeotti *et al.* [17]. This class implements a set of integer values using red-black trees. Each `TreeSet` object has a number of nodes and a pointer to the root node. Each `TreeSetEntry` node stores a value, a color (which can be red or black), and pointers to the left and right children and a parent. The methods for the `TreeSet` class include those to create the empty set, add an element to the set, and remove an element from the set.

Figure 1(b) shows an example of automatically generated tests for the `TreeSet` class. Each test creates an empty set and has a sequence of add and remove operations. The tests are written in the JUnit format [2], but note that these tests have no assertions, i.e., they do not assert that the methods should return certain values. The assumption in this automated generation of test inputs is that outside oracles are used to validate the execution; in the simplest case, one can use a generic oracle that requires each test to terminate regularly, i.e., without throwing an uncaught exception. The goal of generation, hence, is to produce tests that achieve high coverage for some testing criteria.

While the goal of our experiments was to compare the coverage obtained by random testing and shape abstraction, we still found some bugs. For example, using random testing, we found two bugs in the `TreeSet` code from [17]. These bugs resulted in `NullPointerException`s. Note that they were missed by the advanced testing techniques [17] because these techniques did not generate appropriate test inputs.

4 Test Generation for Container Classes

Our aim is to provide more empirical evidence on how random testing compares to shape abstraction in the context of testing containers. To this extent, we generate test suites with the goal to maximize predicate coverage, and also use mutation score for comparison of techniques.

A test suite S consists of n test cases, $S = \{t_1, \ldots, t_n\}$. In general there are different ways to represent and encode a test case. Because we focus on container classes, we use a simple representation that is common in the literature (e.g., [30]): A test case is a sequence of operations such as *add* and *remove* on a container instance created with its default constructor. For the input data, we only consider integer values bounded in $[1,R]$, where R is a fixed constant. The *length* $l(t)$ of a test case t is the number of operations. We do not consider the default constructor in the length. For a test suite S, we define its length as $l(S) = \sum_{t \in S} l(t)$.

4.1 Random Testing

Random testing (RT) is a fast testing technique, in which test cases are simply sampled at random from the input domain. Although RT is often considered a naive testing strategy [20], it can be very effective in many testing scenarios [9, 15]. When the test cases have a variable length representation, there can be different ways to sample test cases at random [9]. However, in this paper we fix the length and number of test cases in each sampled test suite.

Based on the problem definition from Section 4, we analyze the following strategy to generate test suites S. First fix a number n of test cases for S and generate n test cases t with a fixed length $l(t) = k$. The generated test suite S will have length $l(S) = n \times l(t) = nk$. In a random test case, each operation is uniformly chosen (i.e., *add* or *remove*), and the input data is uniformly chosen in $[1,R]$, where R is a constant.

Generating and running a small test suite of n test cases is quite fast for container classes. When the goal is to maximize predicate coverage, an option would be to run RT z times and then output the test suite with highest coverage out of the z runs. How to choose z? This depends on the available testing budget (i.e., for how long a software tester is willing to wait to obtain test data). We can consider two options: (1) run RT for a predefined number of runs z, or (2) run RT several times and stop it after some amount of time (e.g., one second). In practical contexts, option (2) would be preferable and easier to apply. However, option (1) is easier to apply in empirical analyses, because it does not have to deal with the actual execution time (e.g., side effects of implementation/code details, unpredictable delays due to other processes running in parallel, etc.). In this paper, we use only option (1), although we still report indicative times to give a better picture of the techniques' performance.

Once we obtain a test suite of length nk, many method calls might be redundant. Manually verifying the behavior of each operation (e.g., writing assert statements) would likely be too tedious/difficult if no automated oracle is available. Therefore, an approach to deal with this problem is to *minimize* the output test suite S generated by RT, but with the constraint of maintaining the same coverage of the original test suite. We use the following simple minimization algorithm [7]: Remove one method call at

```
1  // inputs: container C, length limit L, values bound R
2  void SA() {
3      Queue<MethodSequence> ToExplore = empty_queue;
4      ToExplore.enqueue(empty_sequence);
5      Set<AbstractState> Explored = empty_set;
6      for (int i = 1; i <= L; i++) {
7          Queue<MethodSequence> NextToExplore = empty_queue;
8          foreach (MethodSequence s: ToExplore) {
9              for (Operation op: {"add", "remove", ...}) {
10                 int[] p = randomPermutationOfRange(1, R);
11                 for (int v: p) {
12                     MethodSequence s' = append(s, op(v));
13                     Container c = create empty C and execute sequence s';
14                     if (execution covered a new predicate combination)
15                         print(s'); // a new test is generated
16                     AbstractState a = abstract(c);
17                     if (a ∉ Explored) {
18                         Explored = Explored ∪ {a};
19                         NextToExplore.enqueue(s'); } } }
20         }
21         ToExplore = NextToExplore; }
22 }
```

Fig. 2. Pseudo-code for shape abstraction (SA) exploration.

a time and re-execute the test case; if the coverage decreases, then re-introduce that method call in the test case. Given a total of nk method calls, this minimization algorithm would require the execution of nk test cases. However, in cases in which we want to make fair comparisons against other techniques, we might want the total length to be at least m function calls. When we minimize a test suite, we can simply stop once the size has reached m.

4.2 Shape Abstraction

The VPP study introduced (explicit execution with abstract matching based on) *shape abstraction* (SA) as a technique for test generation of containers. Unlike RT that produces random sequences of method calls, SA attempts to find that certain sequences are equivalent and hence need not be generated. The original exposition of SA [30] was based on explicit-state model checking, and SA was one of six techniques in the same general framework. We provide a new exposition that directly describes the exploration, focuses solely on SA, and allowed us to obtain a faster implementation of SA without relying on a model checker.

Figure 2 shows the pseudo-code for SA. It takes as input the container code with operations (such as *add* and *remove*), the maximum length of sequences of the operations, and the bounds for the values for those operations. It produces as the output tests (i.e., method sequences) whose execution increases predicate coverage. SA performs a breadth-first search (up to length L) with randomized choices of values (from 1 to R). Line 10 randomly permutes the values to be explored. SA maintains a queue

`ToExplore` of method sequences that still need to be explored and a set `Explored` of *abstract states* that were already encountered.

The key novelty of SA was to compute abstract states using shape abstraction, i.e., ignoring the concrete values in the containers and taking into account only the *shape* in which the container nodes are connected. For example, two red-black `TreeSet` objects that have the same shape of nodes (i.e., the same underlying connection starting from the `root` node and following the `left` and `right` pointers) would map into the same abstract state even if they had different values in those nodes. As a concrete example, consider two balanced trees that each have three nodes and the same red-black colors, one tree with the values 2 in the root, 1 in the left child, and 3 in the right child, and the other tree with the values 4 in the root, 2 in the left child, and 6 in the right child. SA would map these two trees into the same abstract shape.

SA starts the exploration with a queue that has only the empty sequence and with the empty set of abstract states. For each sequence s in the queue (line 8), it randomly chooses an operation and value to apply (lines 9 and 10), extends the sequence to s', executes this sequence[4], prints the sequence if it covered some new predicate combination (lines 14 and 15), and checks if the exploration encountered a new abstract state that should be explored in the future (line 17). Notice that the sequence s' is included in the output test suite whenever its execution increases predicate coverage, even if s' results in an abstract shape that has been already explored and thus s' will not be extended.

5 Case Study

5.1 Subject Containers

Table 2 shows some basic statistics for the 13 subject containers used in our study. For each subject we list a brief identifier, the reference from which we directly obtained the source code, the number of lines of code, the number of mutants generated by the Javalanche mutation tool, and the parameter values for shape abstraction. While we obtained the code directly from three studies [17, 25, 30], all the containers were used previously in many other studies and were originally taken from various sources including Java libraries, textbook implementations done by students, and open source. We included some examples of different implementations of the same containers to see if there are differences in the results.

5.2 Predicate Coverage

Following the VPP study [30], our experiments use a simplified version of the predicate coverage testing criterion. The full predicate coverage, proposed by Ball [11], is a strong criterion that measures how many combinations of *all* program predicates are covered at *all* program points. The predicates are taken from conditional statements and program assertions. For `TreeSet`, for example, the predicates include `t == null`,

[4] The original exposition in VPP [30] assumed a stateful model checker whereas we present SA based on re-execution of method sequences, which does not allow reusing a container from the previous exploration as it may have been modified.

Table 2. Statistics of the subject containers used in our evaluation. For shape abstraction, we set the same value for the length of sequence (L) and the bound for method values (R).

Container	Id	Reference	LOC	Mutants	$L = R$
`AvlTree`	C1	[17]	160	335	20
`BinomialHeap`	C2	[30]	225	289	33
`BinTree`	C3	[30]	94	126	13
`FibHeap`	C4	[30]	245	285	13
`FibonacciHeap`	C5	[25]	319	295	15
`HeapArray`	C6	[25]	75	122	25
`IntAVLTreeMap`	C7	[25]	160	199	20
`IntRedBlackTree`	C8	[25]	228	279	22
`LinkedList`	C9	[17]	176	335	3
`NodeCachingLinkedList`	C10	[17]	172	159	6
`SinglyLinkedList`	C11	[17]	76	167	5
`TreeMap`	C12	[30]	404	651	21
`TreeSet`	C13	[17]	248	360	22

`aKey == t.key`, `t.left != null`, and many others. Unlike the traditional branch, condition, or MCDC coverages [3] that consider values of predicates only *near* where they are used in the code, predicate coverage considers values of predicates at all program points, including *far* from where they are used in the code. Predicate coverage requires using proper variables in scope; for instance, the `remove` method has a variable `TreeSetEntry p`, and predicate coverage would evaluate `p.left != null` (and all other predicates) although there is no such condition in that method.

To make the measurement tractable, the VPP study used only *some* program predicates, and we follow the same approach. However, unlike the VPP study that evaluated predicate coverage at *some* manually selected branches, we use semi-automated instrumentation to evaluate predicate coverage at *all* branches. Our instrumentation is not fully automatic as we manually select the variables for predicates. Describing the predicates and variables we used would be hard, so to enable comparative studies, we made our instrumented code publicly available at http://mir.cs.illinois.edu/coverage.

5.3 Mutation Analysis

Mutation analysis [3] is the process of systematically seeding syntactic changes into a program to determine whether the test cases can detect the resulting semantic program mutants. Undetected ("live") mutants can guide the tester in improving a test suite, while detected ("killed") mutants are used to quantify the effectiveness of a test suite in terms of its *mutation score* that is calculated as the ratio of killed mutants to all mutants.

With appropriate mutation operators, mutation analysis subsumes several traditional coverage criteria such as branch coverage [3]. We are not aware of any study on relationship of mutation analysis and predicate coverage. But an important difference to code coverage is that mutation analysis does not simply check whether some piece of the code has been executed: To (strongly) kill a mutant means to propagate the infected state to an observable output.

We consider output as follows. Given a test case that calls methods on an instance of a container class, we record the state of the container after execution on the original program, and compare it with the state of the container after execution on a mutant program. If there are observable differences in the state, this mutant is considered killed by the test case.

We have implemented this mechanism as an extension to the Javalanche [24] mutation system: Each test case is instrumented automatically with additional instructions that record and compare the state of a container at the end of a test case. To compare states with each other, we simply use the `toString` method, which is commonly overridden by the container classes. In addition, we make sure that all potential instance-specific substrings (e.g., `@` followed by a hexadecimal number) are removed from this output to prevent false positives.

5.4 Experimental Design

For each of the 13 containers, we compare random testing (RT) against shape abstraction (SA). We first run SA, and as shown in Figure 2, it takes two parameters: L is the length of method sequence, and R is the bound for method values. Following the VPP study, we set $L = R$, and we choose the smallest value for L such that (1) the predicate coverage remains constant across 10 different random seeds for L and (2) this predicate coverage is the same for 10 seeds for $L - 1$. The values for L and R are shown in Table 2. We then run the SA experiments for 100 seeds with these bounds.

We run RT as follows. We use 2,000 iterations. For each iteration, a test suite of size $n = 5$ is generated, where each test case has length $k = 200$, so the total length of a test suite is equal to 1,000. Integer inputs are randomly chosen in $[1,R]$, where $R = 20$. After these 2,000 iterations, the test suite with maximum predicate coverage is selected. If several test suites have the same maximum coverage, one test suite is selected at random. This test suite is then minimized, setting the lower bound m for its total length to be the average (across 100 seeds) test suite length obtained with SA on the same container class. (This is the reason why we run SA first.) With successful minimization, this implies that the resulting test suite lengths will be, on average, about the same for both RT and SA. (Note that, in theory, a minimization could even increase predicate coverage while reducing the length of the test case/suite.) Because RT is affected by chance, to obtain enough data to reach reliable conclusions, for each of the 13 containers we ran RT for 100 seeds.

Notice that running RT for more than 2,000 iterations would likely lead to better results because we select from all the iterations one test suite with the highest predicate coverage. For example, we could run RT for the same amount of time that SA takes. However, the problem with doing that would have been the fairness of the comparisons. If two testing techniques (such as RT and SA) are run for the same amount of time, then the worse quality (e.g., measured with predicate coverage) of one technique could be just due to some inefficiencies in the technique's implementation. If a technique has better quality, to increase confidence in the validity of such results, the technique should be also faster. On our machines, running RT for 2,000 iterations takes on average a few seconds, whereas SA is roughly *seven times slower*. In other words, RT consumes less computational resources, and thus its better quality (if any) would have strong validity.

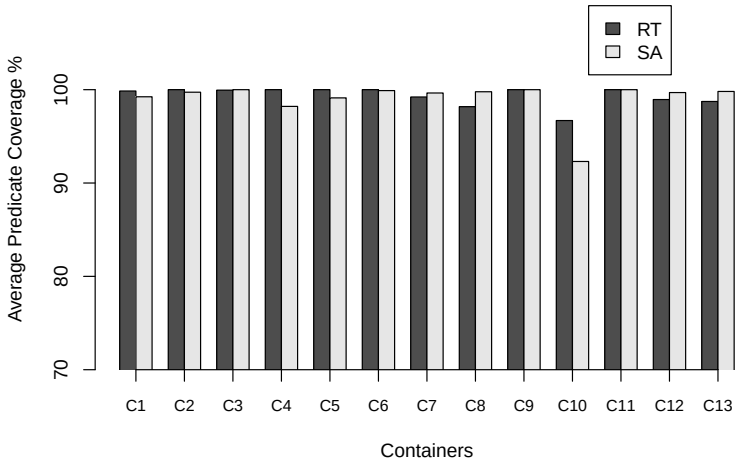

Fig. 3. Average predicate coverage for both random testing (RT) and shape abstraction (SA).

5.5 Results for Predicate Coverage and Mutation Score

Figure 3 shows, for each container, the average predicate coverage divided by the maximum coverage obtained for that container. Specifically, for each container, we first calculated the highest coverage M out of the 200 test suites (100 generated by RT and 100 generated by SA). Then, we divided by M the average predicate coverage for RT (100 test suites) and for SA (100 test suites). The reason for using M is twofold: (1) many predicate combinations could be simply infeasible, and we cannot know how many are feasible, and (2) the number of predicate combinations in various containers is very different, and plotting them without normalizing the data would have led to graphs that are difficult to compare.

Figure 4 presents the results for the mutation analysis, where the average number of killed mutants is reported for each container and testing technique. In contrast to the results for predicate coverage, we did not normalize the data for mutation score. The low mutation score for containers (C1, C3, C4, C9, C10, C11) is partly due to our test generation focusing only on the methods for *add* and *remove* operations, whereas many mutants of these containers are also contained in other methods; likely, extending test generation to include other methods would increase the mutation score. In addition, there is always a number of equivalent mutants which cannot be killed. Because detecting equivalent mutants is an undecidable problem, we included these equivalent mutants in the total number of mutants that was used to calculate the mutation scores.

To analyze these data by taking into account the random components of the techniques, we followed a rigorous statistical procedure [8]. For both comparisons based on predicate coverage and based on mutation score, for each container, when we compare RT against SA, we used a Mann-Whitney U-test to assess whether the effectiveness of these two techniques is statistically different. The resulting *p-values* of these statistical tests indicate the probability of Type I error, i.e., the probability of wrongly stating that there is a difference in quality when actually there is no difference.

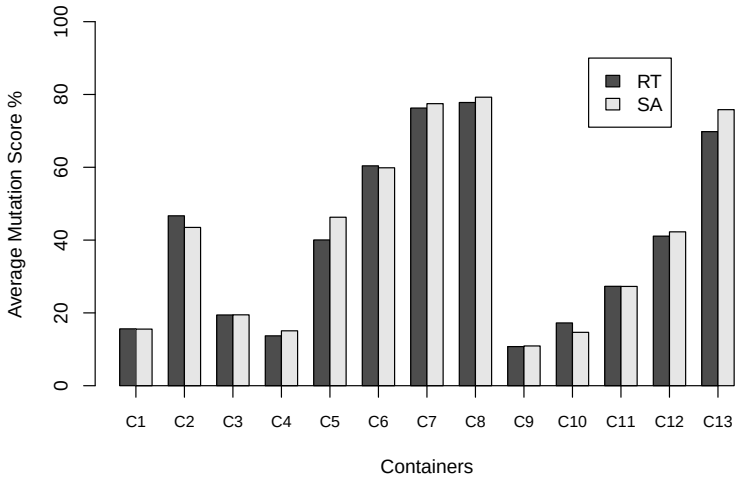

Fig. 4. Average mutation score for both random testing (RT) and shape abstraction (SA).

To assess the magnitude of the difference in a standardized way (i.e., the so called *effect size*), we use the Vargha-Delaney $\hat{A}_{12}$ statistic [28] to compare the quality of RT against SA. In our context, this effect size is an estimate of the probability that a run of RT would give better result than a run of SA. If there is no difference, then we would expect $\hat{A}_{12} = 0.5$. On one hand, if we obtain $\hat{A}_{12} = 1$, this would mean that in *all* the 100 runs of RT we obtained better results than in *all* the 100 runs of SA. On the other hand, if $\hat{A}_{12} = 0$, then it would mean that SA was always better than RT. Table 3 reports the obtained p-values (for the Mann-Whitney U-test) and the $\hat{A}_{12}$ measures (for the Vargha-Delaney statistic).

5.6 Random Testing vs. Shape Abstraction

The experiments show that RT and SA are about equally effective for these 13 containers and the two metrics. For predicate coverage, $\hat{A}_{12} > 0.5$ for five cases, $\hat{A}_{12} < 0.5$ for six cases, and $\hat{A}_{12} = 0.5$ for two cases. For mutation score, $\hat{A}_{12} > 0.5$ for six cases, and $\hat{A}_{12} < 0.5$ for seven cases. Considering the relative behavior of RT against SA, the results for predicate coverage are largely similar to those for mutation score. However, in three of the 13 cases the technique that gives higher predicate coverage does not also give higher mutation score.

Consider first C1. For predicate coverage, the difference is very small ($\hat{A}_{12}$ close to 0.5), and the p-value is rather high, which we can interpret as RT and SA basically behaving similarly. For mutation score, however, the difference is still small ($\hat{A}_{12}$ close to 0.5), but the p-value is rather low, which we can interpret as RT being better than SA for mutation score and thus for C1 overall.

Consider then C4 and C5. They are particularly interesting as RT is always better than SA for predicate coverage, but quite the opposite holds for mutation score. On average, SA achieves 1.38% and 6.24% higher mutation scores for C4 and C5, respectively. Looking at the difference in the sets of mutants killed by SA and RT for these

Table 3. Results of the statistical analysis. The last column shows if random testing is better (RT), shape abstraction is better (SA), both are about equal ($\approx$), or the results are inconclusive ($<>$).

Container	Id	Predicate Coverage		Mutation Score		Better
		p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	Quality
`AvlTree`	C1	0.512	0.487	0.059	0.564	RT
`BinomialHeap`	C2	0.001	0.555	0.001	1.000	RT
`BinTree`	C3	0.001	0.420	0.158	0.490	SA
`FibHeap`	C4	0.001	1.000	0.001	0.191	$<>$
`FibonacciHeap`	C5	0.001	1.000	0.001	0.005	$<>$
`HeapArray`	C6	0.013	0.530	0.001	0.821	RT
`IntAVLTreeMap`	C7	0.001	0.279	0.006	0.388	SA
`IntRedBlackTree`	C8	0.001	0.064	0.001	0.086	SA
`LinkedList`	C9	1.000	0.500	0.514	0.524	$\approx$
`NodeCachingLinkedList`	C10	0.000	0.785	0.001	0.937	RT
`SinglyLinkedList`	C11	1.000	0.500	0.322	0.505	$\approx$
`TreeMap`	C12	0.001	0.144	0.001	0.069	SA
`TreeSet`	C13	0.001	0.076	0.001	0.052	SA

two containers revealed that every single mutant killed by a SA test suite was also killed by at least one of the RT suites. This indicates that RT has a greater variance in mutation score even if it is relatively stable for predicate coverage, which is not surprising as our RT minimization focuses on predicate coverage and not all mutants are directly related to predicates.

Note that C4 and C5 are one example of different implementations of the same data structure. Recall that we intentionally included such implementations among our subjects to evaluate whether the differences between RT and SA depend on the details of the implementations. We find that they largely do not. For example, C4 and C5 behave the same way: RT is better for predicate coverage and SA for mutation coverage. All of C8, C12, and C13 are based on red-black trees, and for all three SA is better than RT (for both predicate coverage and mutation score). C9 and C11 are very similar list implementations, and RT and SA are approximately the same for both. In contrast, C10 is a more complex list implementation, and we find that RT is better than SA (which is consistent with the differences seen for C9 and C11, although those differences are too small to conclude that RT is better). Interestingly, for C1 and C7, which are both based on AVL balanced trees, RT is better than SA for C1, but SA is better than RT for C7.

Recall also that our subjects include not only pointer-based, linked structures (as in the VPP study) but also a container implemented with an array-based structure, namely C6. The results for C6 show that RT is clearly better than SA for this case, but we cannot generalize to all array-based structures.

To summarize, in the context of our study, we cannot identify a superiority of one of the two testing techniques with respect to either predicate coverage or mutation score. However, there is indication that SA performs better for tree-like structures that require complex shapes for coverage (C3, C7, C8, C12, C13), whereas RT performs better for structures that require longer sequences for coverage (C1, C2, C6, C10).

5.7 Bugs

While the goal of our study was to compare RT and SA but not necessarily look for bugs, we still found three real bugs in two containers used in previous studies [17, 25]. Specifically, we found two bugs in the `TreeSet` code from TACO [17] and one bug in the `HeapArray` code from one of our previous studies [25]. The first two bugs led to `NullPointerExpection`s, while the third bug led to an infinite loop. We found all three bugs using RT, and all three bugs were missed by the advanced techniques used in previous studies because those techniques focused on more thorough testing with shorter tests and failed to generate longer tests necessary to reveal these bugs. We reported all three bugs to the original authors of the respective container code, and the authors confirmed them as real bugs and corrected them. The first two bugs were due to a copy-paste mistake, and the third bug was an error of omission.

We also found three bugs that we introduced by mistake in the testing infrastructure that exercised the container code. Specifically, we found one bug in `FibHeap` that resulted in an infinite loop because the test driver was removing a node that did not exist in the structure, and two bugs in `AvlTree` that resulted in `NullPointerExpection`s because our semi-automated instrumentation for measuring predicate coverage changed the original code. We corrected all these bugs, and all our experiments reported above were run with the corrected code.

6 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing tools, we tested them and inspected surprising results. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we ran each experiment 100 times, and we followed rigorous statistical procedures to evaluate their results.

Threats to *construct validity* are on how the quality of a testing technique is defined. We measured not only predicate coverage but also mutation score.

Threats to *external validity* regard the generalization to other types of software, which is common for any empirical analysis. However, in this paper we specifically target container classes and the implementation instances that are commonly used as a benchmark in the literature. The fact that random testing is very efficient in generating effective test cases for container classes will likely not hold for many other types of software. Note that shape abstraction does not apply to all types of software.

7 Conclusion

Containers are important and challenging to test, and many advanced testing techniques were developed for containers. However, there has not been much comparison of these advanced testing techniques with simpler techniques such as random testing. We presented a larger case study that compared random testing with shape abstraction, a state-of-the-art systematic technique. Our experiments showed that random testing achieves comparable results as shape abstraction, but random testing uses much less computation

resources than shape abstraction. We hope that our results provide motivation for future testing studies to (1) compare newly proposed advanced techniques to random testing and/or (2) evaluate newly proposed advanced techniques not (only) on containers but (also) on other code where random testing does not perform well.

Acknowledgments. We thank Marcelo Frias, Juan Pablo Galeotti, Corina Pasareanu, and Willem Visser for providing clarifications about their studies and code used in their experiments. We also thank David Schuler and Andreas Zeller for help with Javalanche. Andrea Arcuri is funded by the Norwegian Research Council. Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. This material is based upon work partially supported by the US National Science Foundation under Grant No. CCF-0746856.

References

1. Jtest. http://www.parasoft.com/jsp/products/jtest.jsp.
2. JUnit. http://http://junit.sourceforge.net/.
3. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
4. J. H. Andrews, A. Groce, M. Weston, and R. G. Xu. Random test run length and effectiveness. In *International Conference on Automated Software Engineering (ASE)*, pages 19–28, 2008.
5. J. H. Andrews, T. Menzies, and F. C. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering (TSE)*, 99(PrePrints), 2010.
6. A. Arcuri. Insight knowledge in search based software testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1649–1656, 2009.
7. A. Arcuri. Longer is better: On the role of test sequence length in software testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 469–478, 2010.
8. A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering (ICSE)*, 2011. (To appear.).
9. A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 219–229, 2010.
10. A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.
11. T. Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects (FMCO)*, pages 1–22, 2005.
12. L. Baresi, P. L. Lanzi, and M. Miraz. TestFul: An evolutionary test approach for Java. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.
13. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(8), 2008.
14. M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

15. J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):438–444, 1984.
16. P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering (TSE)*, 19(8):774–787, 1993.
17. J. Galeotti, N. Rosner, C. López Pombo, and M. Frias. Analysis of invariants for efficient bounded verification. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 25–36, 2010.
18. D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering (TSE)*, 16(12):1402–1411, 1990.
19. K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.
20. G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
21. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.
22. J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51(11):1534–1548, 2009.
23. J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Enabling object reuse on genetic programming-based approaches to object-oriented evolutionary testing. In *European Conference on Genetic Programming (EuroGP)*, pages 220–231, 2010.
24. D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Symposium on The Foundations of Software Engineering (FSE)*, pages 297–298, 2009.
25. R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov. A comparison of constraint-based and sequence-based generation of complex input data structures. In *Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA 2010)*, pages 337–342, 2010.
26. M. Staats and C. Pasareanu. Parallel symbolic execution for structural test generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 183–194, 2010.
27. P. Tonella. Evolutionary testing of classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.
28. A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
29. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, 2004.
30. W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.
31. S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1925–1932, 2006.
32. E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering (TSE)*, 17(7):703–711, 1991.
33. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *International Conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.
34. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, 2005.