Feedback-directed Random Test Generation

Carlos Pacheco[1], Shuvendu K. Lahiri[2], Michael D. Ernst[1], and Thomas Ball[2]
[1]MIT CSAIL, [2]Microsoft Research
{cpacheco,mernst}@csail.mit.edu, {shuvendu,tball}@microsoft.com

Abstract

We present a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created. Our technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts *and* filters*. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.*

Our experimental results indicate that feedback-directed random test generation can outperform systematic and undirected random test generation, in terms of coverage and error detection. On four small but nontrivial data structures (used previously in the literature), our technique achieves higher or equal block and predicate coverage than model checking (with and without abstraction) and undirected random generation. On 14 large, widely-used libraries (comprising 780KLOC), feedback-directed random test generation finds many previously-unknown errors, not found by either model checking or undirected random generation.

1 Introduction

There is an ongoing controversy regarding the relative merits of random testing and systematic testing. Theoretical work suggests that random testing is as effective as systematic techniques [8, 15]. However, some believe that in practice, random testing cannot be as effective as systematic testing because many interesting tests have very little chance of being created at random. Previous empirical studies [9, 18, 28] found that random test input generation achieves less code coverage than systematic generation techniques, including chaining [9], exhaustive generation [18], model checking, and symbolic execution [28].

It is difficult to generalize the results of these studies with regard to the relative advantages of random and systematic testing. The evaluations were performed on very small programs. Because the small programs apparently contained no errors, the comparison was in terms of coverage or rate of mutant killing [21], not in terms of true error detection, which is the best measure to evaluate test input generation techniques. While the systematic techniques used sophisticated heuristics to make them more effective, the type of random testing used for comparison is unguided random testing, with no heuristics to guide its search.

Our work addresses random generation of unit tests for object-oriented programs. Such a test typically consists of a sequence of method calls that create and mutate objects, plus an assertion about the result of a final method call. A test can be built up iteratively by randomly selecting a method or constructor to invoke, using previously-computed values as inputs. It is only sensible to build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception indicating a problem. For example, if the one-method test `a=sqrt(-1)` is erroneous (say, the argument is required to be non-negative), then there is no sense in building upon it to create the two-method test `a=sqrt(-1); b=log(a)`. Our technique uses feedback obtained from executing the sequence as it is being constructed, in order to guide the search toward sequences that yield *new* and *legal* object states. Inputs that create redundant or illegal states are never extended; this has the effect of pruning the search space.

We have implemented the technique in RANDOOP.[1] RANDOOP is fully automatic, requires no input from the user (other than the name of a binary for .NET or a class directory for Java), and scales to realistic applications with hundreds of classes. RANDOOP has found serious errors in widely-deployed commercial and open-source software.

Figure 1 shows a test case generated by RANDOOP when run on Sun's JDK 1.5. The test case shows a violation of the `equals` contract: a set `s1` returned by `unmodifiable-`

[1]RANDOOP stands for "**rand**om tester for **o**bject-**o**riented **p**rograms."

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

Test case for `java.util`

```
public static void test1() {
  LinkedList l1 = new LinkedList();
  Object o1 = new Object();
  l1.addFirst(o1);
  TreeSet t1 = new TreeSet(l1);
  Set s1 = Collections.unmodifiableSet(t1);
  // This assertion fails
  Assert.assertTrue(s1.equals(s1));
}
```

Figure 1. A test case generated by RANDOOP. The test case reveals an error in Sun's JDK 1.5.

`Set(Set)` returns `false` for `s1.equals(s1)`. This violates the reflexivity of `equals` as specified in Sun's API documentation. This test case actually reveals two errors: one in `equals`, and one in the `TreeSet(Collection)` constructor, which failed to throw `ClassCastException` as required by its specification.

Our experimental results indicate that feedback-directed random testing can outperform systematic testing in terms of coverage *and* error detection. On four container data structures used previously to evaluate five different systematic input generation techniques [28], inputs created with feedback-directed random generation achieve equal or higher block and predicate coverage [1] than all the systematic techniques.

In terms of error detection, feedback-directed random testing revealed many errors across 14 widely-deployed, well-tested Java and .NET libraries totaling 780KLOC. Model checking using JPF [26] was not able to create any error-revealing test inputs: the state space for these libraries is enormous, and the model checker ran out of resources after exploring a tiny, localized portion of the state space. Our results suggest that for large libraries, the sparse, global sampling that RANDOOP performs can reveal errors more efficiently than the dense, local sampling that JPF performs. And unlike systematic techniques, feedback-directed random testing does not require a specialized virtual machine, code instrumentation, or the use of constraint solvers or theorem provers. This makes the technique highly scalable: we were able to run RANDOOP on the .NET Framework libraries and three industrial implementations of the JDK, and found previously-unknown errors.

In summary, our experiments indicate that feedback-directed random generation retains the benefits of random testing (scalability, simplicity of implementation), avoids random testing's pitfalls (generation of redundant or meaningless inputs), and is competitive with systematic techniques.

The rest of the paper is structured as follows. Section 2 describes feedback-directed random testing. Section 3 describes experiments that compare the technique with systematic testing and with undirected random testing. Section 4 surveys related work, and Section 5 concludes.

```
public class A {
  public A() {...}
  public B m1(A a1) {...}
}
```

```
public class B {
  public B(int i) {...}
  public void m2(B b, A a) {...}
}
```

sequence s_1	sequence s_2	sequence s_3
`B b1 = new B(0);`	`B b2 = new B(0);`	`A a1 = new A();` `B b3 = a1.m1(a1);`

seqs	*vals*	*extend(*`m2`*, seqs, vals)*

$\langle s_1, s_3 \rangle$	$\langle\ s_1.1, s_1.1, s_3.1\ \rangle$
	(i.e.: `b1`, `b1`, `a1`)
		`B b1 = new B(0);`
		`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`b1.m2(b1,a1);`

$\langle s_3, s_1 \rangle$	$\langle\ s_1.1, s_1.1, s_3.1\ \rangle$
	(i.e.: `b1`, `b1`, `a1`)
		`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`B b1 = new B(0);`
		`b1.m2(b1,a1);`

$\langle s_1, s_2 \rangle$	$\langle\ s_1.1, s_2.1, \texttt{null}\ \rangle$
	(i.e.: `b1`, `b2`, `null`)
		`B b1 = new B(0);`
		`B b2 = new B(0);`
		`b1.m2(b2,null);`

Figure 2. Three example applications of the *extend* operator.

2 Technique

An object-oriented unit test consists of a sequence of method calls that set up state (such as creating and mutating objects), and an assertion about the result of the final call. This section describes a randomized, feedback-directed technique for generating such unit tests.

2.1 Method Sequences

A *method sequence*, or simply *sequence*, is a sequence of method calls. Each call in the sequence includes a method name and input arguments, which can be primitive values (i.e., constants like `0`, `true` or `null`) or reference values returned by previous method calls. (We treat the receiver, if any, as the first input argument.) We write $s.i$ to mean the value returned by the i-th method call in sequence s. This notation applies only to non-void methods.

When giving the textual representation of a sequence, we print it as code and assign identifiers (names) to return values of method calls. This is only for ease of understanding; specific identifiers are not part of a sequence and are only necessary when outputting a sequence as code.

2.2 Extending sequences

This section defines an extension operation that takes zero or more sequences and produces a new sequence. Extension is the core operation of the feedback-directed generation algorithm. The extension operation creates a new sequence by concatenating its input sequences and appending a method call at the end. More formally, the operator *extend(m, seqs, vals)* takes three inputs:

- ***m*** is a method with formal parameters (including the receiver, if any) of type $T_1, \ldots, T_k$.
- ***seqs*** is a list of sequences.

GenerateSequences(*classes*, *contracts*, *filters*, *timeLimit*)

```
1  errorSeqs ← {} // Their execution violates a contract.
2  nonErrorSeqs ← {} // Their execution violates no contract.
3  while timeLimit not reached do
4    // Create new sequence.
5    m(T_1 ... T_k) ← randomPublicMethod(classes)
6    ⟨seqs, vals⟩ ← randomSeqsAndVals(nonErrorSeqs, T_1 ... T_k)
7    newSeq ← extend(m, seqs, vals)
8    // Discard duplicates.
9    if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
10     continue
11   end if
12   // Execute new sequence and check contracts.
13   ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
14   // Classify new sequence and outputs.
15   if violated = true then
16     errorSeqs ← errorSeqs ∪ {newSeq}
17   else
18     nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
19     setExtensibleFlags(newSeq, filters, o⃗)  // Apply filters.
20   end if
21 end while
22 return ⟨nonErrorSeqs, errorSeqs⟩
```

Figure 3. Feedback-directed generation algorithm for sequences.

- ***vals*** is a list of values $v_1 : T_1, \ldots, v_k : T_k$. Each value is a primitive value, or it is the return value $s.i$ of the i-th method call for a sequence s appearing in *seqs*.

The result of *extend(m, seqs, vals)* is a new sequence that is the concatenation of the input sequences *seqs* in the order that they appear, followed by the method call $m(v_1, \ldots, v_k)$. Figure 2 shows three examples of applying the operator. Both reuse of a value (as illustrated in the first example) and use of distinct duplicate sequences (as illustrated in the third example) are possible.

2.3 Feedback-directed generation

Figure 3 shows the feedback-directed random generation algorithm. It builds sequences incrementally, starting from an empty set of sequences. As soon as a sequence is built, it is executed to ensure that it creates non-redundant and legal objects, as specified by *filters* and *contracts*. The algorithm takes four inputs: a list of *classes* for which to create sequences, a list of *contracts*, a list of *filters*, and a time limit (*timeLimit*) after which the generation process stops. RANDOOP provides default contracts, filters, and time limit (2 minutes), so the only required argument is the list of classes.

A sequence has an associated boolean vector: every value $s.i$ has a boolean flag $s.i$.*extensible* that indicates whether the given value may be used as an input to a new method call. The flags are used to prune the search space: the generator sets a value's *extensible* flag to *false* if the value is considered redundant or illegal for the purpose of creating a new sequence. Section 2.4 explains how these flags are set.

Sequence creation first selects a method $m(T_1 \ldots T_k)$ at random among the public methods of *classes* (line 5). Next, it tries to apply the extension operator to m. Recall that the operator also requires a list of sequences and a list of values; the helper function *randomSeqsAndVals*$(T_1 \ldots T_k)$ (called on line 6 of Figure 3) incrementally builds a list of sequences *seqs* and a list of values *vals*. At each step, it adds a value to *vals*, and potentially also a sequence to *seqs*. For each input argument of type T_i, it does the following:

- If T_i is a primitive type, select a primitive value from a fixed pool of values. (In the implementation, the primitive pool contains a small set of primitives like `-1`, `0`, `1`, `'a'`, `true`, etc., and can be augmented by the user or by other tools.)
- If T_i is a reference type, there are three possibilities: use a value v from a sequence that is already in *seqs*; select a (possibly duplicate) sequence from *nonErrorSeqs*, add it to *seqs*, and use a value from it; or use `null`. The algorithm selects among these possibilities at random. (By default, it uses `null` only if no sequence in *nonErrorSeqs* produces a value of type T_i.) *When using a value v of type T_i produced by an existing sequence, the value must be extensible, that is, $v.extensible = true$.*

The sequence *newSeq* is the result of applying the extension operator to *m*, *seqs*, and *vals* (line 7). The algorithm checks whether an *equivalent* sequence was already created in a previous step (lines 9–11). Two sequences are equivalent if they translate to the same code, modulo variable names. If *newSeq* is equivalent to a sequence in *nonErrorSeqs* or *errorSeqs*, the algorithm tries again to create a new sequence.

Now, the algorithm has created a new (i.e. not previously-created) sequence. The helper function $execute(newSeq, contracts)$ executes each method call in the sequence and checks the contracts after *each* call. In other words, the contracts express invariant properties that hold both at entry and exit from a call. A contract takes as input the current state of the system (the runtime values created in the sequence so far, and any exception thrown by the last call), and returns *satisfied* or *violated*. (This terminology differs from some other uses of "contract" in the literature.) Figure 4 shows the default contracts that RANDOOP checks.

The output of *execute* is the pair $\langle \vec{\mathbf{o}}, violated \rangle$ consisting of the runtime values created during the execution of the sequence,[2] and a boolean flag *violated*. The flag is set to true if at least one contract was violated during execution. A sequence that leads to a contract violation is added to the set *errorSeqs* (lines 15 to 16). If the sequence leads to no contract violations, line 18 adds it to *nonErrorSeqs*, and line 19 applies filters to it (see Section 2.4).

[2] We use a bold sans-serif font for variables that hold runtime values of the classes under test.

Method	
contract	**description**
Exception (Java)	method throws no `NullPointerException` if no input parameter was null
	method throws no `AssertionError`
Exception (.NET)	method throws no `NullReferenceException` if no input parameter was null
	method throws no `IndexOutOfRangeException`
	method throws no `AssertionError`
Object	
contract	**description**
equals	`o.equals(o)` returns `true`
	`o.equals(o)` throws no exception
hashCode	`o.hashCode()` throws no exception
toString	`o.toString()` throws no exception

Figure 4. Default contracts checked by RANDOOP. Users can extend these with additional contracts, including domain-specific ones. A contract is created programmatically by implementing a public interface.

RANDOOP outputs the two input sets *nonErrorSeqs* and *errorSeqs* as JUnit/NUnit tests, along with assertions representing the contracts checked. The first set contains sequences that violate no contracts and are considered non-redundant and legal with respect to the filters given. These are tests that the tested classes pass; they could be used for regression testing. The second set contains sequences that violate one or more contracts. These are tests that the classes fail; they indicate likely errors in the code under test.

2.4 Filtering

Line 19 of Figure 3 applies *filters* (given as inputs to the algorithm) that determine which values of a sequence are extensible and should be used as inputs to new method calls. A filter takes as input a sequence and the values resulting from its execution. As a result of applying a filter to a sequence s, the filter may set some $s.i.extensible$ flags to *false*, with the effect that the value will not be used as input to new method calls. The helper function $setExtensibleFlags(newSeq, filters, \vec{\mathbf{o}})$ in line 19 iterates through the list of filters given as input to the algorithm and applies each filter to *newSeq* in turn. Below we describe the three filters that RANDOOP uses by default.

Equality. This filter uses the `equals()` method to determine if the resulting object has been created before. The filter maintains a set **allobjs** of all extensible objects that have been created by the algorithm across all sequence executions (the set can include primitive values, which are boxed). For each value $newSeq.i$ in the sequence, it sets $newSeq.i.extensible$ to *false* if the runtime **o** corresponding to $newSeq.i$ is such that $\exists$**o'** $\in$ **allobjs** : **o**.`equals(`**o'**`)`.

This heuristic prunes any object with the same abstract value as a previously-created value, even if their concrete representations differ. This might cause RANDOOP to miss an error, if method calls on them might behave differently. The heuristic works well in practice but can be disabled or refined by the user. For instance, it is straightforward to use reflection to write a method that determines whether two objects have the same concrete representation (the same values for all their fields), or a user could specify more sophisticated computations to determine object equality [30].

Null. Null dereference exceptions caused by using `null` as an argument are often uninteresting, and usually point to the (possibly intentional) absence of a null check on the arguments. However, when a null dereference exception occurs in the absence of any null value in the input, it often indicates some internal problem with the method. The null filter sets $newSeq.i.extensible$ to *false* iff the corresponding object is `null`.

Null arguments are hard to detect statically because the arguments to a method in a sequence themselves are outputs of other sequences. Instead, the null filter checks the values computed by execution of a specific sequence.

Exceptions. Exceptions frequently correspond to precondition violations for a method, and therefore there is little point extending them. Furthermore, an extension of the sequence would lead to an exception before the execution completes. This filter prevents the addition of a sequence to the *nonErrorSeqs* set if its execution leads to an exception, even if the exception was not a contract violation.

2.5 Repetition

Sometimes, a good test case needs to call a given method multiple times. For example, repeated calls to `add` may be necessary to reach code that increases the capacity of a container object, or repeated calls may be required to create two equivalent objects that can cause a method like `equals` to go down certain branches. To increase the chances that such cases are reached, we build repetition directly into the generator, as follows. When generating a new sequence, with probability N, instead of appending a single call of a chosen method m to create a new sequence, the generator appends M calls, where M is chosen uniformly at random between 0 and some upper limit max. (max and N are user-settable; the default values are $max = 100$ and $N = 0.1$.) There are other possible ways to add repetition to the generator (e.g., we could repeat parameters or entire subsequences).

3 Evaluation

This section presents the results of three experiments that evaluate the effectiveness of feedback-directed random input generation. Section 3.1 evaluates the coverage that RANDOOP achieves on a collection of container data structures, and compares it with that achieved by systematic input generation techniques implemented in the JPF model checker [26, 28]. Section 3.2 uses RANDOOP to generate test inputs that find API contract violations on

		coverage				time (seconds)			
		JPF	RP	JPF_U	RP_U	JPF	RP	JPF_U	RP_U
block coverage	BinTree	.78	.78	.78	.78	0.14	0.21	0.14	0.13
	BHeap	.95	.95	.95	.86	4.3	0.59	6.2	6.6
	FibHeap	1	1	1	.98	23	0.63	1.1	27
	TreeMap	.72	.72	.72	.68	0.65	0.84	1.5	1.9
predicate coverage	BinTree	53.2	54	52.1	53.9	0.41	1.6	2.0	4.2
	BHeap	101	101	88.3	58.5	9.8	4.2	12	15
	FibHeap	93	96	86	90.3	95	6.0	16	67
	TreeMap	106	106	104	55	47	10	10	1.9

JPF : Best-performing of 5 systematic techniques in JPF.
RP : RANDOOP: Feedback-directed random testing.
JPF_U : Undirected random testing implemented in JPF.
RP_U : Undirected random testing implemented in RANDOOP.

Figure 5. Basic block coverage (ratio) and predicate coverage (absolute) achieved by four input generation techniques.

14 widely-used libraries, and compares with JPF and with undirected random testing (as implemented in RANDOOP and in JCrasher [3]). Section 3.3 uses RANDOOP-generated regression test cases to find regression errors in three industrial implementations of the Java JDK.

3.1 Generating test inputs for containers

Container classes have been used to evaluate many input generation techniques [18, 31, 30, 27, 28]. In a recent paper [28], Visser et al. compared basic block and a form of predicate coverage [1] achieved by several input generation techniques on four container classes: a binary tree (`BinTree`, 154 LOC), a binomial heap (`BHeap`, 355 LOC), a fibonacci heap (`FibHeap`, 286 LOC), and a red-black tree (`TreeMap`, 580 LOC). They used a form of predicate coverage that measures the coverage of all combinations of a set of predicates manually derived from conditions in the source code. They compared the coverage achieved by six techniques: (1) model checking, (2) model checking with state matching, (3) model checking with abstract state matching, (4) symbolic execution, (5) symbolic execution with abstract state matching, and (6) undirected random generation.

Visser et al. report that the technique that achieved highest coverage was model checking with abstract state matching, where the abstract state records the shape of the container and discards the data stored in the container. For brevity, we'll refer to this technique as *shape abstraction.* Shape abstraction dominated all other systematic techniques in the experiment: it achieved higher coverage, or achieved the same coverage in lesser time, than every other technique.[3] We compared feedback-directed random generation with shape abstraction. For each data structure, we performed the following steps.

1. We reproduced Visser et al.'s results for shape abstraction on our machine (Pentium 4, 3.6GHz, 4G memory, running Debian Linux). We used the optimal parameters reported in [28] (i.e., the parameters for which the technique achieves highest coverage).
2. We ran RANDOOP on the containers, specifying the same methods under test as [28]. Random testing has no obvious stopping criterion; we ran RANDOOP for two minutes (its default time limit).
3. To compare against unguided random generation, we also reproduced Visser et al.'s results for random generation, using the same stopping criterion as [28]: generation stops after 1000 inputs.
4. To obtain a second data point for unguided random generation, we ran RANDOOP a second time, turning off all filters and heuristics.

As each tool ran, we tracked the coverage achieved by the test inputs generated so far. Every time a new unit of coverage (basic block or predicate) was achieved, we recorded the coverage and time. To record coverage, we reused Visser et al.'s experimental infrastructure, with small modifications to track time for each new coverage unit. For basic block coverage, we report the ratio of coverage achieved to maximum coverage possible. For predicate coverage, we report (like Visser et al. [28]) only absolute coverage, because the maximum predicate coverage is not known. We repeated each run ten times with different seeds, and report averages.

Figure 5 shows the results. For each ⟨ technique, container ⟩ pair, we report the maximum coverage achieved, and the time at which maximum coverage was reached, as tracked by the experimental framework. In other words, the time shown in Figure 5 represents the *time that the technique required in order to achieve its maximum coverage*—after that time, no more coverage was achieved in the run of the tool. (But the tool may have continued running until it reached its stopping criterion: on average, each run of JPF with shape abstraction took a total of 89 seconds; the longest run was 220 seconds, for `TreeMap`. Every run of RANDOOP took 120 seconds, its default time limit.)

For `BinTree`, `BHeap` and `TreeMap`, feedback-directed random generation achieved the same block and predicate coverage as shape abstraction. For `FibHeap`, feedback-directed random generation achieved the same block coverage, but higher predicate coverage (96 predicates) than shape abstraction (93 predicates). Undirected random testing was competitive with the other techniques in achieving block coverage. For the more challenging predicate coverage, both implementations of undirected random testing always achieved less predicate coverage than feedback-directed random generation.

[3] Random generation was able to achieve the same predicate coverage as shape abstraction in less time, but this happened only for 2 (out of 520) "lucky" runs.

We should note that the container data structures are non-trivial. For `BHeap`, to achieve the highest observed block coverage, a sequence of length 14 is required [28]. This suggests that feedback-directed random generation is effective in generating complex test inputs—on these data structures, it is competitive with systematic techniques.

`FibHeap` and `BHeap` have a larger input space than `BinTree` and `TreeMap` (Visser et al. defined more testable methods for them, which leads to more possible sequences). It is interesting that for `FibHeap` and `BHeap`, feedback-directed random generation achieved equal or greater predicate coverage as shape abstraction, and did so faster (2.3 times faster for `BHeap` and 15.8 times faster for `FibHeap`), despite the higher complexity. This suggests that feedback-directed random generation is competitive with systematic generation even when the state space is larger. (The observation holds for much larger programs used in Section 3.2).

Another interesting fact is that repetition of method calls (Section 2.5) was crucial. When we analyzed the inputs created by feedback-directed random generation, we saw that for `FibHeap` and `TreeMap`, sequences that consisted of several element additions in a row, followed by several removals, reached predicates that were not reached by sequences that interleaved additions with removals. This is why undirected random generation achieved less coverage.

Two other systematic techniques that generate method sequences for containers are Rostra [30] and Symstra [31]. Rostra generates tests using bounded exhaustive generation with state matching. Symstra generates method sequences using symbolic execution and prunes the state space based on symbolic state comparison. Unfortunately, the tools were not available to us. The authors of Rostra and Symstra remarked [29] that for evaluation purposes, their techniques are comparable with those evaluated by Visser et al.

The best measure to evaluate input generation techniques is error detection, not coverage. Our results suggest that further experimentation is required to better understand how systematic and random techniques compare in detecting errors in data structures. The next section evaluates feedback-directed random generation's error-detection ability on widely-used libraries, and compares it with systematic and (unguided) random generation.

3.2 Checking API contracts

In this experiment, we used feedback-directed random generation, undirected random generation, and systematic generation to create test suites for 14 widely-used libraries comprising a total of 780KLOC (Figure 6). Section 3.2.1 describes the results for feedback-directed random testing. Section 3.2.2 describes the results for systematic testing. Section 3.2.3 describes the results for undirected random testing.

To reduce the amount of test cases we had to inspect, we implemented a test runner called REDUCE, which can replace JUnit or NUnit. Like those tools, REDUCE shows only failing tests, but REDUCE only shows a subset of the failing tests. REDUCE partitions the failing tests into equivalence classes, where two tests fall into the same class if their execution leads to a contract violation after the same method call. For example, two tests that exhibit a contract failure after a call to the JDK method `unmodifiableSet(Set)` belong to the same equivalence class. This step retains only one test per equivalence class (chosen at random); the remaining tests are discarded.

Java libraries	**LOC**	**public classes**	**public methods**	description
Java JDK 1.5				
java.util	39K	204	1019	Collections, text, formatting, etc.
javax.xml	14K	68	437	XML processing.
Jakarta Commons				
chain	8K	59	226	API for process flows.
collections	61K	402	2412	Extensions to the JDK collections.
jelly	14K	99	724	XML scripting and processing.
logging	4K	9	140	Event-logging facility.
math	21K	111	910	Mathematics and statistics.
primitives	6K	294	1908	Type-safe collections of primitives.
.NET libraries	**LOC**	**public classes**	**public methods**	
ZedGraph	33K	125	3096	Creates plots and charts.
.NET Framework				
Mscorlib	185K	1439	17763	.NET Framework SDK class libraries.
System.Data	196K	648	11529	Provide access to system functionality
System.Security	9K	128	1175	and designed as foundation on which
System.Xml	150K	686	9914	.NET applications, components, and
Web.Services	42K	304	2527	controls are built.

Figure 6. Libraries used for evaluation.

3.2.1 Feedback-directed random generation

For each library, we performed the following steps:

1. We ran RANDOOP on a library, specifying all the public classes as targets for testing. We used RANDOOP's default parameters (contracts from Figure 4, filters from Section 2.4, and 2 minute time limit). The output of this step was a test suite.
2. We compiled the test suite and ran it with REDUCE.
3. We manually inspected the failing test cases reported by REDUCE.

For each iteration, we report the following statistics.

1. **Test cases generated.** The size of the test suite (number of unit tests) output by RANDOOP.
2. **Violation-inducing test cases.** The number of violation-inducing test cases output by RANDOOP.
3. **REDUCE reported test cases.** The number of violation-inducing test cases reported by REDUCE (after reduction and minimization) when run on the RANDOOP-generated test suite.
4. **Error-revealing test cases.** The number of test cases reported by REDUCE that revealed an error in the library. We made this determination as follows.

library	test cases generated	violation-inducing test cases	REDUCE reported test cases	error-revealing test cases	errors	errors per KLOC
Java JDK						
java.util	22,474	298	20	19	6	.15
javax.xml	15,311	315	12	10	2	.14
Jakarta Commons						
chain	35,766	1226	20	0	0	0
collections	16,740	188	67	25	4	.07
jelly	18,846	1484	78	0	0	0
logging	764	0	0	0	0	0
math	3,049	27	9	4	2	.09
primitives	49,789	119	13	0	0	0
ZedGraph	8,175	15	13	4	4	.12
.NET Framework						
Mscorlib	5,685	51	19	19	19	.10
System.Data	8,026	177	92	92	92	.47
System.Security	3,793	135	25	25	25	2.7
System.Xml	12,144	19	15	15	15	.10
Web.Services	7,941	146	41	41	41	.98
Total	208,503	4200	424	254	210	

Figure 7. Statistics for test cases generated by RANDOOP. Section 3.2.1 explains the metrics.

Java libraries. We labeled a test case as error-revealing only if it violated an explicitly stated property in the documentation for the code in question.

.NET libraries. The design guidelines for .NET require that public methods respect the contracts in Figure 4 (i.e. .NET has a stronger specification). We labeled each distinct method that violated a contract as an error for the .NET programs: a method that leads to the contract violation either contains an error, fails to do proper argument checking, or fails to prevent internal errors from escaping to the user of the library. Because REDUCE reports one test case per such method, REDUCE-reported test cases coincide with error-revealing test cases for .NET.

5. **Errors.** The number of distinct errors uncovered by the error-revealing test cases. We count two errors as distinct if fixing them would involve modifying different source code.
6. **Errors per KLOC.** The number of distinct errors divided by the KLOC count for the library.

Errors discovered. Figure 7 shows the results. RANDOOP created a total of 4200 distinct violation-inducing test cases. Of those, REDUCE reported approximately 10% (and discarded the rest as potentially redundant). Out of the 424 tests that REDUCE reported, 254 were error-revealing. The other 170 were illegal uses of the libraries or cases where the contract violations were documented as normal operation. The 254 error-revealing test cases pointed to 210 distinct errors. Next we give representative examples of the errors (for more details, see the longer technical report [24]).

Eight other methods in the JDK create collections that return `false` on `s.equals(s)` (like Figure 1). These eight methods shared some code, and together they revealed four distinct errors. In the Jakarta libraries, a matrix class's implementation of `hashCode` fails to handle a valid object configuration where a specific field is `null`. In a different error, an iterator object throws a `NullPointerException` if initialized with zero elements (also a valid configuration). In the .NET libraries, 155 errors are `NullReferenceException`s in the absence of `null` inputs, 21 are `IndexOutOfRangeException`s, and 20 are violations of `equals`, `hashCode` or `toString` contracts. RANDOOP also led us to discover nonterminating behavior in System.Xml. This error was assigned the highest priority ranking (it can render unusable an application) and was fixed almost immediately.

3.2.2 Systematic Testing

To compare feedback-directed random testing with systematic testing, we used JPF to test the Java libraries. JPF does not actually create method sequences—to make it explore method sequences, the user has to manually write a driver program that nondeterministically calls methods of the classes under test, and JPF explores method sequences by exploring the driver (for instance, Visser et al. wrote driver programs for the container experiments [28]). We wrote a *universal driver* generator which, given a set of classes, creates a driver that explores all possible method sequences up to some sequence length, using only public methods and constructors. For this experiment, we augmented the drivers with the code that checked the same contracts as RANDOOP (Figure 4). We performed the experiments on a Pentium 4, 3.6GHz, 4G memory, running Debian Linux.

For each library, we generated a universal driver and had JPF explore the driver until it ran out of memory. We specified sequence length 10 (this was greater than the length required to find all the Java errors from Figure 7). We used JPF's breadth-first search strategy, as done for all systematic techniques in [28]. In that paper, Visser et al. suggest that BFS is preferable than DFS for this kind of exploration scenario. We used JPF's default state matching (shape abstraction is not currently implemented in JPF, other than for the four containers from Section 3.1).

For all the libraries, JPF ran out of memory (after 32 seconds on average) without reporting any errors. Considering the size of the libraries, this is not surprising, as JPF was barely able to explore the libraries before state space explosion became a problem.

RANDOOP was able to explore the space more effectively not because it explored a larger portion of the state space—it only explored a tiny fraction of an enormous state space. For example, java.util declares about 1000 public methods; consider how many sequences of length 10 are possible. While JPF thoroughly sampled a tiny, localized portion of the space, RANDOOP sparsely sampled a larger portion. Our results suggest that for large libraries,

sparse, global sampling can reveal errors more efficiently than dense, local sampling.

jCUTE [25] performs *concolic testing*, a systematic technique that performs symbolic execution but uses randomly-generated test inputs to initialize the search and to allow the tool to make progress when symbolic execution fails due to limitations of the symbolic approach (e.g. native calls). Comparing feedback-directed random generation with concolic testing would be interesting. Unfortunately, jCUTE crashed when compiling the drivers generated for the classes because it could not handle drivers of the size generated for our subject programs.

3.2.3 Undirected Random Testing

To measure the benefits of feedback-directed random testing versus undirected random testing, we reran RANDOOP as described in Section 3.2.1 a second time, using the same parameters, but disabling the user of filters or contracts to guide generation. Across all libraries, unguided generation created 1,326 violation-inducing test cases. Out of these, REDUCE reported 60 test cases, all of which pointed to distinct errors (58 in the .NET libraries, and 2 in the Java libraries). Undirected generation did not find any errors in java.util or javax.xml, and was unable to create the sequence that uncovered the infinite loop in System.Xml (to confirm that this was not due simply to an unlucky random seed, we ran RANDOOP multiple times using different seeds; undirected generation never found the bug).

JCrasher [3] is an independent implementation of undirected random test generation whose goal is to uncover exceptional behavior that points to an error. JCrasher randomly generates tests, then removes tests that throw exceptions not considered by JCrasher to be potentially fault-revealing. We used JCrasher to generate test cases for the Java libraries. JCrasher takes as input a list of classes to test and a "depth" parameter that limits the number of method calls it chains together. We ran JCrasher with maximum possible depth.

JCrasher ran for 639 seconds, created a total of 698 failing test cases, of which 3 were error-revealing and revealed one distinct error (using the same counting methodology as in Section 3.2.1). Jcrasher created many redundant and illegal inputs that could be detected using feedback-directed heuristics. See [24] for a detailed description of the test cases.

Recent work has introduced a new tool, Check 'n' Crash [4], that improves JCrasher by replacing its random generation by constraint solving. It would be interesting to compare this technique to ours, or to combine their strengths.

3.3 Regression and compliance testing

This section describes a case study in which we used feedback-directed random testing to find inconsistencies between different implementations of the same API. As our subject program, we used the Java JDK. We tested three commercial implementations: Sun 1.5, Sun 1.6 beta 2, and IBM 1.5. The goal was to discover inconsistencies between the libraries which could point to regression errors in Sun 1.6 beta 2 or compliance errors in either of the libraries. RANDOOP can optionally create a *regression oracle* for each input, which records the runtime behavior of the program under test on the input by invoking observer methods on the objects created by the input. RANDOOP guesses observer methods using a simple strategy: a method is an observer if all of the following hold: (i) it has no parameters, (ii) it is public and non-static, (iii) it returns values of primitive type (or `String`), and (iv) its name is `size`, `count`, `length`, `toString`, or begins with `get` or `is`.

We ran RANDOOP on Sun 1.5, using the option that creates regression oracles and the default time limit. RANDOOP generated 41,046 regression test cases. We ran the resulting test suite using Sun 1.6 beta and a second time using IBM 1.5. A total of 25 test cases failed on Sun 1.6, and 73 test cases failed on IBM 1.5. On inspection, 44 out of the 98 test cases revealed inconsistencies that uncovered 12 distinct errors in the implementations (other inconsistencies reflected different implementations of a permissive specification). See [24] for the specific inconsistencies.

All distributed JDKs must pass an extensive compliance test suite (`https://jck.dev.java.net/`, regrettably not available to the public nor to us). Nevertheless, RANDOOP was able to find errors undiscovered by that suite. Internally, IBM extensively uses comparisons against the Sun JDK during testing, but they estimate that it will take 100 person-years to complete that comparative testing [16]. A tool like RANDOOP could provide some automated support in that process.

4 Related Work

Automatic test input generation is an active research area with a rich literature. We focus on input generation techniques that create method sequences.

Input space representation. Techniques that generate method sequences must first describe what a method sequence is. Despite the apparent simplicity of such a task, previous representations are not expressive enough to describe all method sequences that can be created for a set of classes.

Rostra [30] and Symstra [31] internally use Henkel and Diwan's *term-based* representation [17, 29]. For example, the term `pop(push(s,i).state)` is equivalent to the sequence `s.push(i); s.pop()`. This representation cannot express reuse of an object (aliasing): the sequence `Foo f = new Foo(); f.equals(f)` is not expressible as a term. The representation also cannot express mutation of an object via a method that mutates its parame-

ter: the sequence `List l = new ArrayList(); ...; Collections.shuffle(l); l.add(2)` is not expressible as a term. While not explicitly stated, JCrasher [3] and Eclat [23] follow an equivalent representation and thus suffer from the same lack of expressiveness.

Random testing. Random testing [14] has been used to find errors in many applications; a partial list includes Unix utilities [19], Windows GUI applications [10], Haskell programs [2], and Java programs [3, 23, 22].

JCrasher [3] creates test inputs by using a "parameter graph" to find method calls whose return values can serve as input parameters. RANDOOP does not explicitly create a parameter graph; instead it uses a component set of previously-created sequences to find input parameters. RANDOOP creates fewer redundant and illegal inputs because it discards component sequences that create redundant objects or throw exceptions. JCrasher creates every input from scratch and does not use execution feedback.

Feedback-directed test generation was introduced with the Eclat tool [23] (developed by two authors of this paper). Like RANDOOP, Eclat creates tests that are likely to expose errors by performing random generation augmented by automatic pruning based on execution results. Eclat prunes sequences that appear to be illegal because they make the program behave differently than a set of correct training runs. The previous work focused on automatic classification of tests in the absence of an oracle.

The present work presents an orthogonal set of techniques that focus on generating a set of behaviorally-diverse test inputs, including state matching to prune redundant objects, repetition to generate low-likelihood sequences, oracles based on API contracts that can be extended by the user, and regression oracles that capture the behavior of a program when run on the generated input. Eclat's performance is sensitive to the quality of the sample execution given as an input to the tool. Since RANDOOP does not require a sample execution, it is not sensitive to this parameter. Finally, the present work contributes a number of experimental evaluations, including a comparison with JPF, a widely-used tool. An experimental comparison of Eclat and RANDOOP (or their combination) is an interesting avenue for future work, as it could help understand the strengths and weaknesses of different feedback-directed approaches.

Systematic testing. Many techniques have been proposed to systematically explore method sequences [30, 4, 31, 12, 25, 5, 28]. Bounded exhaustive generation has been implemented in tools like Rostra [30] and JPF [28]. JPF and Rostra share the use of state matching on objects that are receivers of a method call, and prune sequences that create a redundant receiver. RANDOOP performs state matching on values other than the receiver and introduces the finer-grained concept of a sequence that creates some redundant and some nonredundant objects (using a boolean flag for each object in the sequence). Only sequences that create nothing but redundant objects are discarded. Rostra and JPF do not favor repetition or use contracts during generation to prune illegal sequences or create oracles. Rostra is evaluated on a set of 11 small programs (34–1000 LOC), and JPF's sequence generation techniques were evaluated on 4 data structures; neither tool found errors in the tested programs.

An alternative to bounded exhaustive exploration is symbolic execution, implemented in tools like Symstra [31], XRT [12], JPF[26], and jCUTE [25]. Symbolic execution executes method sequences with symbolic input parameters, builds path constraints on the parameters, and solves the constraints to create actual test inputs with concrete parameters.

Check-n-Crash [4] creates abstract constraints over inputs that cause exceptional behavior, and uses a constraint solver to derive concrete test inputs that exhibit the behavior. DSD [5] augments Check-n-Crash with a dynamic analysis to filter out illegal input parameters.

Combining random and systematic. Ferguson and Korel [9] proposed an input generation technique that begins by executing the program under test with a random input, and systematically modifies the input so that it follows a different path. Recent work by Godefroid et al [11, 25] explores DART, a symbolic execution approach that integrates random input generation. RANDOOP is closer to the other side of the random-systematic spectrum: it is primarily a random input generator, but uses techniques that impose some systematization in the search to make it more effective. Our approach and more systematic approaches represent different tradeoffs of completeness and scalability, and thus complement each other.

Comparing random and systematic. Theoretical studies have shown that random testing is as effective as more systematic techniques such as partition testing [15, 20]. However, the literature contains relatively few empirical comparisons of random testing and systematic testing. Ferguson and Korel compared basic block coverage achieved by inputs generated using their chaining technique versus randomly generated inputs [9]. Marinov et al. [18] compared mutant killing rate achieved by a set of exhaustively-generated test inputs with a randomly-selected subset of inputs. Visser et al. [28] compared basic block and a form of predicate coverage achieved by model checking, symbolic execution, and random testing. In all three studies, undirected random testing achieved less coverage or killed fewer mutants than the systematic techniques.

In previous work [6], we compared Eclat's random generation and classification techniques [23] with Symclat, a symbolic version of Eclat. We conjectured that random generation may benefit from using repetition; this was the motivation for implementing repetition in RANDOOP.

5 Conclusion

Feedback-directed random testing scales to large systems, quickly finds errors in heavily-tested, widely-deployed applications, and achieves behavioral coverage on par with systematic techniques.

The exchange of ideas between the random and systematic approaches could benefit both communities. Groce et al. propose structural heuristics [13] to guide a model checker; the heuristics might also help a random test generator. Going the other way, our notion of exploration using a component set, or state matching when the universe contains more than one object, could be translated into the exhaustive testing domain. Combining random and systematic approaches can result in techniques that retain the best of each approach.

Acknowledgments. We thank Willem Visser for sharing his subject programs and experimental framework and answering questions about JPF. We also thank David Glasser and Adam Kieżun for running and analyzing the results of the JDK regression experiment (Section 3.3).

References

[1] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO*, pages 1–22, 2004.

[2] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, Sept. 2000.

[3] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[4] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431, May 2005.

[5] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *ISSTA*, pages 245–254, July 2006.

[6] M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE*, Sept. 2006.

[7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 4(11):34–41, Apr. 1978.

[8] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE TSE*, 10(4):438–444, July 1984.

[9] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM TOSEM*, 5(1):63–86, Jan. 1996.

[10] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, pages 59–68, Seattle, WA, USA, Aug. 2000.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, June 2005.

[12] W. Grieskamp, N. Tillmann, C. Campbell, W. Schulte, and M. Veanes. Action machines – towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *QSIC 2005: Quality Software International Conference*, Sept. 2005.

[13] A. Groce and W. Visser. Heuristics for model checking Java programs. *STTT*, 6(4):260–276, 2004.

[14] D. Hamlet. Random testing. In *Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[15] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE TSE*, 16(12):1402–1411, Dec. 1990.

[16] A. Hartman. Personal communication, July 2006.

[17] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*, pages 431–456, July 2003.

[18] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT/LCS/TR-921, MIT Lab for Computer Science, Sept. 2003.

[19] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *CACM*, 33(12):32–44, Dec. 1990.

[20] S. Ntafos. On random and partition testing. In *ISSTA*, pages 42–48, Mar. 1998.

[21] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, Oct. 2000.

[22] C. Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, pages 242–256, Sept. 2005.

[23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, July 2005.

[24] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Sept. 2006.

[25] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, Aug. 2006.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *ASE*, 10(2):203–232, 2003.

[27] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, July 2004.

[28] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, July 2006.

[29] T. Xie. Personal communication, Aug. 2006.

[30] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, pages 196–205, Nov. 2004.

[31] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, Apr. 2005.