Randoop: Feedback-Directed Random Testing for Java

Carlos Pacheco   Michael D. Ernst

MIT CSAIL

{cpacheco,mernst}@csail.mit.edu

Abstract

RANDOOP FOR JAVA generates unit tests for Java code using feedback-directed random test generation. Below we describe RANDOOP's input, output, and test generation algorithm. We also give an overview of RANDOOP's annotation-based interface for specifying configuration parameters that affect RANDOOP's behavior and output.

Categories and Subject Descriptors: D.2.5 SOFTWARE ENGINEERING: Testing and Debugging

General Terms: Verification.

Keywords: Java, random testing.

1. Introduction

Unit testing is an important and widely-practiced activity in software development. Unfortunately, writing unit tests is often tedious, difficult and time consuming. This paper describes RANDOOP FOR JAVA, a tool that automatically generates random but meaningful unit tests for Java code (we have also developed a .NET version of RANDOOP, used internally at Microsoft). RANDOOP is available at

    http://people.csail.mit.edu/cpacheco/randoop/

RANDOOP takes as input a set of classes under test, a time limit, and optionally, a set of "contract checkers" that extend those used by RANDOOP as default. RANDOOP outputs two tests suites. One contains *contract-violating tests* that exhibit scenarios where the code under test leads to the violation of an API contract. For example, `test1` in Figure 1 is a contract-violating test that reveals an error in Java's collections framework. It shows a way of creating a set `s` that violates reflexivity of equality in Sun's JDK 1.5 and 1.6. In particular, the call `s.equals(s)` returns `false`, which violates the API for `java.lang.Object`. RANDOOP implements a de-
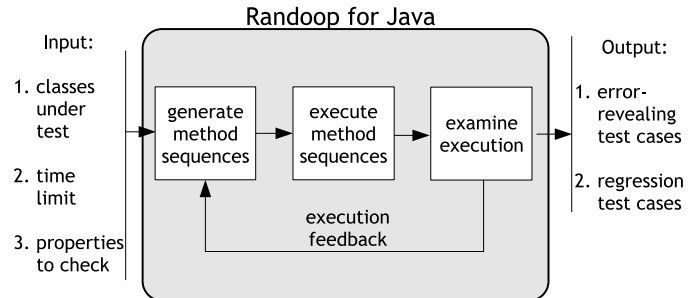
Example error-revealing test

```
// Fails on Sun 1.5, 1.6.
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);
    Assert.assertTrue(s1.equals(s1));
}
```

Example regression test

```
// Passes on Sun 1.5, fails on Sun 1.6 Beta 2.
public static void test2() {
    BitSet b = new BitSet();
    Assert.assertEquals(64, b.size());
    b.clone();
    Assert.assertEquals(64, b.size());
}
```

Figure 1. RANDOOP's input is a set of classes, a time limit, and optionally, a set of contract checkers. It outputs contract-violating tests and regression tests.

fault set of contracts (including reflexivity of equality) that the user can extend.

The second suite that RANDOOP outputs contains *regression tests,* which do not violate contracts but instead capture an aspect of the current implementation. For example, `test2` in Figure 1 shows a regression test that records an aspect of method `BitSet.size()`'s behavior (specifically, cloning a `BitSet` does not change its size). Regression tests can discover inconsistencies between two different versions of the software; `test2` reveals an inconsistency between Sun's JDK 1.5 (on which the test was generated, and thus succeeds by construction) and Sun's JDK 1.6 Beta 2 (on which the test fails). Contract-violating tests and regression tests both have as their purpose finding errors—the former in the current im-

plementation of the classes under test, and the latter in future or different implementations.

2. How it works

Randoop generates unit tests using *feedback-directed random testing,* a technique inspired by random testing that uses execution feedback gathered from executing test inputs as they are created, to avoid generating redundant and illegal inputs [5]. RANDOOP creates method sequences incrementally, by randomly selecting a method call to apply and selecting arguments from previously constructed sequences.

As soon as it is created, a new sequence is executed and checked against a set of contracts. Sequences that lead to contract violations are output to the user as contract-violating tests. Sequences that exhibit normal behavior (no exceptions and no contract violations) are output as regression tests. Finally, sequences that exhibit illegal behavior (for example, a sequence that throws an `IllegalArgumentException`) are discarded. Only normally-behaving sequences are used to generate new sequences, as it makes little sense to extend a sequence that contains an already-corrupted state.

In [5] we showed that feedback-directed random test generation can outperform both systematic and undirected random test generation, in terms of coverage and error detection. On four small but nontrivial data structures, RANDOOP achieved higher or equal block and predicate coverage than model checking and undirected random generation. On 14 large, widely-used libraries (comprising 780KLOC), RANDOOP found many previously-unknown errors not found by either model checking or undirected random generation.

3. Writing contracts

To specify a contract to be checked, the user declares a class implementing a *contract-checking interface*. For example, to specify a contract involving a single object, the user declares a class implementing `randoop.UnaryObjectChecker`. The crucial method in this interface is `boolean check(Object)`, which returns `true` iff the given object satisfies the contract. RANDOOP defines additional interfaces to create other types of contracts, for example, contracts involving more than one object (e.g. if `o1.equals(o2) == true` then `o1.hashCode() == o2.hashCode()`), or contracts specifying the expected effect of a method on the program state.

4. Annotations

In addition to writing contracts, the user can annotate methods to direct RANDOOP's test generation algorithm. To date, we have implemented three kinds of annotations, all at the method level.

- `@Omit` indicates that the given method should not be used to generate tests. For example, methods that exhibit non-deterministic behavior (e.g. `System.currentTimeMillis()`) are typically not good candidates for RANDOOP-based

generation, because they can cause generated tests to pass or fail unpredictably from one run to the next.

- `@Observer` indicates that a method should be used to create regression assertions. For example, the method `BitSet.size()` from the regression test in Figure 1 is an observer method.

- `@ObjectInvariant` indicates that a parameterless method that returns a boolean value is an object invariant. Object invariant methods are used similarly to contract checkers: if an object invariant method returns `false` the test input is considered to be error-revealing and output to the user with an appropriate assertion.

5. Related work

Tools for generating unit tests for Java code include JCrasher [2], Eclat [4], and Jartege [3] (academic), and Jtest [6] and Agitator [1] (commercial). JCrasher creates sequences of method calls for Java programs and reports sequences that throw certain types of exceptions. Jartege lets the user specify properties to check as formal specifications written in JML [3]. This contrasts with RANDOOP's use of Java interfaces and annotations to specify properties to check. The philosophy behind our choice is that a tool will be more useful to practicing programmers if they can express contracts using constructs they already know, rather than learning a new formal language. (JML is more expressive than our contracts, so our choice also represents a tradeoff in expressiveness.) Neither JCrasher nor Jartege generate regression tests, support annotations, or use runtime feedback, which can lead the tools to generate many redundant and illegal tests that RANDOOP can avoid.

In previous work, we developed the Eclat [4] test generation tool. like RANDOOP, Eclat uses execution feedback to guide its generation. Instead of user-provided contracts, Eclat uses a predefined set of dynamically-inferred properties (and requires the user to provide an existing test suite from which to infer these properties). RANDOOP does not automatically infer properties, so it does not require a pre-existing test suite. Instead, it uses a set of universally-applicable object properties, and lets the user extend this set to express properties specific to the implementation under test.

References

[1] Agitar. http://www.agitar.com.

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

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

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

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[6] Parasoft Corporation. *Jtest*. http://www.parasoft.com/.