Scaling Up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs

Brian Robinson
ABB Corporate Research
brian.p.robinson@us.abb.com

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Jeff H. Perkins
MIT CSAIL
jhp@csail.mit.edu

Vinay Augustine
ABB Corporate Research
vinay.augustine@us.abb.com

Nuo Li
ABB Robotics
nuo.li@cn.abb.com

***Abstract*—This paper presents an automatic technique for generating maintainable regression unit tests for programs. We found previous test generation techniques inadequate for two main reasons. First. they were designed for and evaluated upon libraries rather than applications. Second, they were designed to find bugs rather than to create maintainable regression test suites: the test suites that they generated were brittle and hard to understand. This paper presents a suite of techniques that address these problems by enhancing an existing unit test generation system. In experiments using an industrial system, the generated tests achieved good coverage and mutation kill score, were readable by the product's developers, and required few edits as the system under test evolved. While our evaluation is in the context of one test generator, we are aware of many research systems that suffer similar limitations, so our approach and observations are more generally relevant.**

I. Introduction

The benefits of unit and regression testing are widely recognized. These tests reveal defects that would be difficult and expensive to find and fix in later development phases, verify that changes to the code do not break existing functionality, and are integral to practices such as test-driven development and continuous integration. A key obstacle to adopting unit testing is the costs to *develop* and to *maintain* a unit test suite.

The cost to manually *develop* a regression unit test suite for an existing codebase — most of which have no such tests — is usually prohibitive. In addition, if the original developers are not involved in creating the tests, the team creating the tests must determine the correct behavior of the code on their own. And, developers are often more interested in (and better rewarded for) developing functionality than test code. An alternative to creating an entire regression test suite is to incrementally create unit tests for new code as it is developed. This fails to address the bulk of the existing codebase, and regression defects often make up a significant percentage of the defects detected in a new release. A recent internal study of a product at ABB found that 80% of post-release defects over the last three years were either latent defects injected many releases previously, or regression defects due to changes in the code. For this product, creating unit tests for only new code would provide very little perceived benefit.

The cost to *maintain* a regression unit test suite is also a problem. As the software evolves over time, existing tests must be updated to reflect changes in behavior or in the product's specification. There is a tradeoff between sensitivity and brittleness: a test suite that detects many erroneous changes is also likely to issue false positive failures for many desired changes.[1] A developer must update or remove each failing test, and if updating is difficult or tedious, the temptation to remove it becomes too great to resist. Ten years ago, a product development team at ABB focused on creating a good automated unit test suite. At substantial cost, they created a suite that achieved ~90% statement coverage. But, maintaining that suite was too costly, due to brittle and difficult-to-understand test cases, and it decayed over time. Today, it has ~10% statement coverage and is rarely run. This example shows that the coverage of a test suite alone does not make it successful, but rather the balance between coverage and maintainability.

The costs of developing and maintaining tests are not perceived to justify the benefits of unit testing. As a result, adequate adoption of unit testing by large industrial software development organizations remains low [24]. Our goal in this work is to tip the balance of the cost-benefit tradeoff, by automatically generating *maintainable regression tests* for *real software programs*.

Existing test generation techniques (such as random or concolic testing) have limitations that prevent them from fulfilling either part of the goal (*generate maintainable regression tests* or *real software programs*). For example, existing tools focus on detecting crashes or exceptions. While this is helpful in finding new defects, a *maintainable regression suite* needs test oracles that determine if a new version of the software behaves differently than the previous version. In addition, existing tools have been evaluated on library code [20], [5], [22], [21], [19] rather than *real software programs*. As we further describe in Section III, libraries are easier for tools to handle and do not contain many of the complexities that real software programs have. For example, real software programs contain persistent state and make use of external code, which prevent existing concolic testing techniques from working. Finally, to be industrially practical, a test generation technique should

[1] An example false positive failure is requiring a new version's output to be identical to the previous version's output, even though the program's specification permits multiple behaviors, such as allowing iteration through a set in an arbitrary order.

work without human interaction and without assuming a formal specification or test oracle.

Since existing techniques do not meet our goal, we decided to extend a test generation technique. We chose Randoop (an implementation of feedback-directed random testing) as the most practical available test generation tool. We did not use symbolic/concolic techniques (such as Java PathFinder) because of their limitations when applied to real systems [23]. Our ideas would be equally applicable to any attempt to scale up such techniques to generate regression tests for real programs.

This paper makes two main contributions.

- We present a suite of enhancements to the feedback-directed random testing approach, and to the Randoop tool. These enhancements address the challenge of creating effective, maintainable regression tests for real software programs. The enhancements are publicly available at http://randoop.googlecode.com/. Each enhancement is conceptually simple but is sometimes subtle in its application, and their combination is novel.
- We evaluated our extensions in the context of a large, mature industrial software system. The generated tests have good code coverage and mutation score — better than the manually-written tests that the development team runs each week. Although the generated tests cover much of the code and are sensitive to many faults (and even exposed a previously unknown bug), they are easy for a developer to maintain, requiring just 4 edits to accommodate 2 years and 3 public versions of changes.

The rest of this paper is organized as follows. Section II reviews the feedback-directed random test generation technique, and Section III describes how we extended it. Section IV explains our experimental methodology, and Section V presents the results. Section VI compares our research to related work, and Section VII states our conclusions and identifies future work.

II. Randoop

Our work extends the Randoop tool, which automatically generates unit tests. Randoop implements a technique called feedback-directed random testing [20], [22].

In feedback-directed random test generation, a test is built up iteratively. Each iteration randomly selects a method or constructor to invoke, using previously computed values as inputs. The 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. The key idea of feedback-directed testing is to execute each test as soon as it is generated, and to use information gathered from that execution to bias the test generation process as it creates further tests. The bias makes the test generator less likely to generate illegal tests, and less likely to generate redundant tests. In particular, inputs that create redundant or illegal states are never extended, which has the effect of pruning the search space.

An object-oriented unit test consists of a sequence of method calls that set up state (such as creating and mutating objects), and at least one assertion about the result of the final call. Randoop creates such sequences iteratively: it creates test cases out of short sequences, then extends the sequences to create larger test cases.

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

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

Pool of previously-constructed sequences		
`B b1 = new B(0);`	`B b2 = new B(0);`	`A a1 = new A();` `B b3 = a1.m1(a1);`

3 possible extensions		

`B b1 = new B(0);`
`A a1 = new A();`
`B b3 = a1.m1(a1);`
`b1.m2(b1,a1);`
	`A a1 = new A();`
	`B b3 = a1.m1(a1);`
	`B b1 = new B(0);`
	`b1.m2(b1,a1);`
		`B b1 = new B(0);`
		`B b2 = new B(0);`
		`b1.m2(b2,null);`

Fig. 1. Example code, and 6 method call sequences that could be used in the body of a unit test for the code. The bottom three sequences show three possible extensions, if Randoop chooses to extend sequences in the pool so as to make a sequence that ends with `m2`. Adapted from [22].

To create a new test case, Randoop needs to choose a method to test, as well as arguments and assertions. See Figure 1 for an illustration.

- To choose the method, Randoop uses random choice among the public methods of the classes under test.
- To aid in choosing the arguments, Randoop maintains a pool of all values that have been generated by any test case so far, along with the sequence of calls that created the object. Randoop initializes the pool with a set of values such as `null`, `-1`, `0`, `1`, `'a'`, `true`, etc. For each argument to the chosen method, Randoop selects a random value, of the proper type, from the pool.
- Randoop creates assertions that detect errors. Randoop can optionally create a regression test that detects deviations from previous behavior by adding assertions about the final state of the object after the test completes.
  To detect errors, Randoop uses the following built-in set of assertions. No method should throw `AssertionError`. No method should throw `NullPointerException` if none of its arguments was null. Some methods, such as `equals`, `hashCode`, and `toString`, should never throw an error. For any object, `o.equals(o)` should return true. A user may extend these built-in assertions with domain-specific properties, but this is not required.

III. Enhancements to Randoop

Randoop was originally targeted toward detecting existing bugs in data structure libraries libraries such as the JDK's `java.util`. Instead, we want to extend Randoop to generate maintainable regression tests for complex industrial software systems.

Data structure libraries tend to be easier for tools to handle in several ways.

1) Most library methods are deterministic. Non-deterministic results tend to fall into simple patterns, such as the default return value of `toString`, which

prints a unique identifier similar to a hash code. There is little or no global state; global state can give the appearance of non-determinism.

2) Most utility library methods do not make changes to the underlying system or the outside world (such as databases). As a result, the methods can be called in any order, and most classes do not have to be initialized before they can be used.
3) Most utility library methods do not interact with the external environment. By contrast, a program's code may query the user for information, exit the program, etc. However, this behavior is not acceptable in an automated regression test. Furthermore, libraries undergo few code changes and even fewer behavior changes to existing functionality. By contrast, the test suite of an evolving program may need to be updated frequently during development.
4) Because the library is written generically, simple arguments can exercise all of its behavior, and that behavior depends on only simple properties of the arguments, such as equality and ordering. By contrast, a program manipulates much more complex data, and correctness properties may depend on part or all of that data.
5) A library is less likely to be in need of a regression test suite. Libraries rarely change, and a library is likely to already have some tests. Test generation is used chiefly to find bugs (as opposed to creating regression tests), possibly using a partial specification of permitted behavior. Only failing tests are shown to the user, while succeeding tests can be discarded without examination. Readability, maintainability, and redundancy of tests generated for a library are secondary concerns.
6) General-purpose libraries are less likely than programs to have specific "magic constants" that must be used in order to exercise certain behaviors.

The following six sections describe enhancements we made to extend Randoop to to create maintainable regression tests for real software programs. The first two changes are needed by any technique to create robust, behavior-preserving test oracles. The next two are changes needed by any technique to support testing real software programs. The final two changes are enhancements specific to Randoop that improve its generated tests, no matter what context Randoop is used it. All of these enhancements are publicly available and documented at http://randoop.googlecode.com/.

A. Remove Non-deterministic Observations

A method can return different results on different invocations. This can happen because of nondeterminism (concurrency, the current time, dependence on hash codes, the memory management system, etc.), but is more often caused by dependence on program state that is exposed by a different ordering of calls. Randoop needs to ensure that its tests do not depend on the results of (apparently) non-deterministic methods, because such a test would produce many false positive failures.

A concrete example from our case study is a static variable of type `Vector` Randoop calls `Vector` methods such as `add()`, `remove()`, and `size()` on its value, and the order of such calls affects their return values. This is not just a problem in Randoop-generated tests; manually-written test suites also have data dependences that make results change if tests are run in a different order [16].

During test generation, Randoop executes each statement in a test and records certain outcomes as assertions. These assertions may fail when the test suit is run. As described in Section II, Randoop discards possible method sequences as it is running that lead to redundant object states, exceptions (also discussed in Section IV-A), or other error conditions. These sequences may have modified the global state in ways that other generation-time observations depended upon.

Randoop previously addressed the problem of test outcome nondeterminism using a very simplistic approach: it discarded (did not place in the pool) `String` values that contained the pattern typical of `Object.toString()`. This was sufficient for certain libraries on which Randoop had been evaluated, but not for real programs.

We developed a new approach. Now, after generating an entire test suite, Randoop runs it before outputting it. If any test fails, Randoop assumes the failing test is nondeterministic and retains the test in the suite but disables each failing assertion.

Alternative approach: An alternative approach we considered would remove any non-deterministic method call from the test suite. This approach would yield a smaller test suite, since the suite does not contain calls whose results are ignored. This simple approach does not work, because removing a method call in one test can cause a subsequent, putatively independent, test to fail. This is because tests can interact through shared variables: side effects in one test cause differences to the behavior of subsequent tests. Even changing the *order* of tests in a suite (or, equivalently, performing test case selection or prioritization on the suite) can cause other tests to fail. For example, if an array is smaller than it was on a previous test execution, a test that previously succeeded could now throw a index-out-of-bounds exception. In conclusion, (apparently) non-deterministic behavior cannot be removed from a suite, as its removal could easily lead to other changes in the global state and further spuriously-failing tests.

B. Prevent Changes to the Underlying System

Application code may depend on and make changes to the underlying system (e.g., add, delete, or modify files in the file system). Putting aside for the moment the fact that this may lead to nondeterministic behavior, automatically created unit tests may perform unexpected or even catastrophic changes to the computer system.

To prevent this problem, Randoop now uses a Java security manager. A security manager can prevent certain operations, and is useful in contexts beyond preventing security breaches. In our case study, the security manager gives permissions to specific properties and the socket used to connect to the database.

A second, related change is that Randoop now provides an option for its user to specify setup code that should be performed at the beginning of testing or of running a test. This was necessary in our case study, as the program connects to a database and performs other initialization activities.

C. Modify Inappropriate Calls

Application code may interact with the user (such as creating a dialog box) in a variety of different situations. In our case study, which targeted the logic connected to a GUI, this was often only to confirm a particular action. If such calls occur during either test generation or test execution, the tests cannot be run without manual intervention.

We added a feature to Randoop that allows a user to specify a mapping from current method calls to a replacement call. For example, the `javax.swing.JOptionPane.showMessageDialog` method, which usually presents a dialog box, can be replaced with a call that simply prints out the message and returns. The feature works via bytecode rewriting. In the experiments, we used this feature to remove dialog boxes that require a response. We also removed calls to `System.exit()`.

We found another compelling use for this feature. During test maintenance, it can be used to understand regression test errors that result from behavioral changes. A developer can selectively (on a per-method basis) revert application behavior back to the behavior in the *previous* software release, in order to reproduce old behavior, cause a failing regression test to succeed again, and thereby verify exactly what changes caused a regression test failure.

For example, in our case study we discovered that, between versions, the developers changed the behavior of a method so that it created two columns in a database rather than one. This simple change caused a relatively large number of failures in the automatically-generated regression unit tests. We mapped each of these calls to a wrapper routine that converted to the old arguments and results. The complete code required was 11 lines long.

Once the regression tests have been verified on a new release (including any possible changes to the new version due to errors that were uncovered), a tester has two choices. The tester can update the suite, or the tester can simply discard it and use Randoop to build a new regression suite for the new version. The wrapper routines are not a permanent fixture of the tests but are only used when debugging failing tests on a newer software version.

D. Observe Pure Methods

As described in Section II, Randoop tests that a result has the expected value. For example:

```
assert result.equals(expected);
```

Using an existing `equals` method may be too strict (because `equals` checks undesired fields or data, or even implements reference equality) or too lenient (because `equals` skips important information).

We modified Randoop to apply a set of user-defined observer methods to the value and check their results. An observer method is a method with no side effects. Thus, instead of having a single assertion at the end of a generated test, there may be many assertions at the end, one for each applicable observer method. For example:

```
assert result.f1().equals(expected.f1());
assert result.f2().equals(expected.f2());
assert result.f3().equals(expected.f3());
```

We define an observer method as a pure nullary non-`void` method; that is, it has no side effects, it takes no arguments except the receiver, and it returns a value. For instance, a getter method such as `Point.getX()` is an observer method. Randoop takes as input a set of observer methods; a user could mark these manually, or use an automated analysis to compute them.

Randoop utilizes observer methods in a second way: to avoid making no-op method calls. Randoop conservatively assumes that any method may side-effect any of its arguments. Without this assumption, Randoop would never call a `void` method! Another way of saying this is that Randoop treats each method as having multiple outputs: its return value and the final values of its non-primitive arguments. However, this can lead to useless calls in the middle of a sequence, if the call has no side effect. Randoop can use knowledge about (lack of) side effects to ignore some of these possible outputs. Randoop's equality testing can often detect lack of side effects, but our approach is more direct and efficient.

E. Filter Lexically Redundant Tests

Randoop builds larger tests out of smaller ones. This means that every time Randoop adds a test, the new test lexically subsumes at least one, and perhaps many, shorter tests. For example, suppose that methods `a()` through `d()` perform side effects. Randoop might output these four tests:

```
{ a(); }
{ a(); b(); }
{ a(); b(); c(); }
{ a(); b(); c(); d(); }
```

The first three tests are lexically redundant. As a heuristic, we enhanced Randoop to remove them from the test suite that it generates.

If Randoop is run with a time limit, then removing lexically redundant tests creates smaller and easier-to-understand suites. If Randoop is run with a size goal (number of tests to output), then removing lexically redundant tests rather than outputting them has three effects. Generation time increases slightly. The test run time also increases marginally, since the tests tend to be slightly larger on average (the smaller ones were removed). Most importantly, test suite quality improves, because the resulting suite has more diversity than it would have had otherwise.

Removing lexically redundant tests is a heuristic, because the removed tests are not necessarily semantically redundant. In the example above, it is possible that `a()` has internal state, and that a fault occurs only after calling it four times (also see Section III-A). In this case, removing the first three tests

would reduce the fault detection capability of the overall test suite. We have never observed this effect in practice.

Even though the redundant tests are removed from the generated test suite, they are kept in the pool, to permit creating, for example, `{ a(); b(); e(); }`.

We implemented this feature as a response to ABB developers' objections to the original Randoop-generated tests. When industrial developers examined the tests, they identified, and objected to, lexical redundancies in the tests. They thought the redundancies might negatively impact maintainability, by making some tests needlessly longer and by increasing the number of redundant test failures when the software evolves.

F. Use Source Code Literals as Arguments

A constant that appears in the source program may be relevant to its behavior. For example, a program may have different behavior when a data structure is larger vs. smaller than a given threshold. A parser may expect a string to start with a given sequence of characters; the parser may not exhibit interesting behavior for malformed inputs. Test code may define a special user and password, without which certain functionality cannot be accessed. An integer may be used as an enumeration to control behavior.

Randoop previously used only a small set of constant values: for numbers, -1, 0, 1, 10, and 100; for characters, `'#'`, `' '`, `'4'`, and `'a'`; and for strings, `""` and `"hi!"`. This may be adequate to exercise much of a utility library, but is inadequate to exercise a real program.

We enhanced Randoop to add, to the initial pool, any constants that appear in the source code under test. Suppose a constant appears in class `package.Class`. As a user option, Randoop can use the constant only when generating tests within class `package.Class`, for any tests in package `package`, or anywhere. Our experiments use the package choice. Using the source code literals indiscriminately was not effective, because a realistic program has many literals, and the pool would be so large that the likelihood of choosing the right one for any particular method would be vanishingly small. Using the class granularity was suboptimal as well: we saw cases in which a literal in one class needed to be supplied as as argument to a method in a different class (that might eventually call the first one).

IV. Study Design

We conducted a study with two main goals.

1. Determine the impact of our Randoop enhancements (Section III) on the *quality* of the generated tests. To determine the improvement due to each added feature, we generated test suites with and without the added features enabled. As a measure of quality, we used statement coverage and mutation kill score (see Section IV-D). These two metrics are believed to be correlated with defect detection.

2. Determine how *maintainable* the Randoop-generated regression test suite is. To measure this, we used Randoop to create a test suite for one version of a system, ran the test suite on a subsequent version of the system, and determined how much effort was required to return the suite to the passing state.

When generating tests, we used all of Randoop's defaults, including its maximum of 100 seconds of test generation time per class.

Sections IV-A and IV-B expand on our methodology for assessing quality and maintainability, respectively.

A. Measuring Quality

We refined the quality goal into these research questions:

1) Can Randoop create an effective regression suite for a non-library program?
2) Does using string literals from the program improve the effectiveness of the generated tests?
3) Does removing lexically redundant tests improve the effectiveness of the generated test suites?
4) Does using observer methods improve the effectiveness of the tests generated?
5) Which has greater impact on test effectiveness, larger test suites or longer test cases?

The results for these research questions are shown in Section V-A.

We evaluated the first research question by measuring the quality of Randoop-generated tests and by comparing them to good-quality human-generated tests. We evaluated the next three research questions by running Randoop to generate two test suites, one with and one without the given feature, and comparing the suites. We used a test suite size of 2000 tests per class — beyond this size, additional tests have negligible effect. We evaluated the fifth research question by generating and comparing test suites of different sizes.

We evaluated each pair of suites based on their *statement coverage* and *mutation kill score*. (We also computed branch coverage, but the results were similar, so this paper omits those measurements for brevity.) We computed statement coverage for all classes, and computed mutation scores for 72 classes (see Section IV-D for justification).

It would be better to directly measure defect detection: the number of previously-unknown errors that each test suite reveals. This is infeasible to compute for most real software systems: the lack of a formal specification makes it difficult to know whether a given test execution exposes a failure. For example, most exceptions thrown by randomly-generated tests are valid behavior caused by illegal inputs. To avoid a flood of false positives, we ran Randoop in a mode where it ignored any test that throws an exception. Randoop thus created regression tests that ensured the software's non-exceptional behavior did not change.

B. Measuring Maintainability

We refined the maintainability goal into these research questions:

1) How much editing is required to maintain the generated tests through a major version change?
2) Does removing lexically redundant tests improve the maintainability of the generated test suites?

3) What is developers' opinion of the readability and maintainability of the generated tests?

The results for these research questions are shown in Section V-B.

We evaluated test maintainability by using the generated unit test suite as a regression test suite: that is, we ran it on a later version of the software. Any failures on this later version are due to a behavioral change in the software — either a regression defect or an intended change. A test failure due to intended behavioral changes is seen as a false positive by developers. There is a tradeoff between the sensitivity of a test suite and its brittleness. Ideally, a test suite should detect many defects, but should issue few false positives when the code changes over time. A suite that misses too many defects is useless for testing, and a suite that issues too many hard-to-interpret false positives will be too hard to maintain and will be abandoned by developers.

Similar to many other test generation strategies, Randoop has many parameters that affect the test suites that it generates. We measured the effect of three parameters that we believed, based on our previous experience, would be important: test suite size, test case size, and test suite run time. In addition to a quantitative analysis of test suite maintenance, we asked the developers to evaluate the generated test suite qualitatively, as the acceptance of the test suite by the development teams is critical.

C. Industrial System Studied

Our experiments use a subject program that we call Rata. Rata is a mature real-time monitoring and control product developed at ABB. Rata contains two parts of approximately equal size. The control algorithms and connectivity components in the system are written in C. The business logic, database connectivity, and UI are written in Java, and consist of 652 total classes. Of these 652 classes, only 568 are testable, as 41 classes are interfaces and 43 are GUI components. The 568 testable classes are the focus of this study and contain 127k lines of code.

Rata has above-average quality when compared to other ABB products, with regards to field defects. The product does not currently have a unit-level regression test suite. Release testing currently finds the majority of the defects. The development team would like to have a unit test suite, but there is no business case for manually writing one. The Rata development team does have a black-box regression suite they run manually each week. We call this the "manual suite".

D. Mutation Analysis

The mutation kill score is computed by mutation analysis. Mutation analysis constructs many variants of a program, each of which differs from the original in some small respect, such as by replacing one instance of "+" by "-". The variants are called "mutants", and the code changes are called "mutations". Given a set of mutants, a test suite's mutation score, or kill score, is the fraction of mutants that are detected by the test suite. A mutation score is computed by the following equation: # killed mutants / # total mutants. A killed mutant is a mutation that is detected by a test in the suite. The main purpose of mutation is comparing two test suites: the suite with the larger score is better.

Creating mutants for all 568 classes of Rata and evaluating all of the generated test suites against them would be computationally infeasible. Therefore, we asked the Rata developers to identify the package they see as the most critical. We performed mutation analysis on only this package. (Randoop works on all of Rata, and our other results use all 568 testable classes.)

We created mutants using MuJava [13] version 3 and the MuClipse plugin [26] for the Eclipse IDE. However, MuJava does not work on classes that contain GUI components. It created mutants for 72 of the 109 classes in the package. Therefore, our mutation results in this paper are for those 72 classes.

We now describe our configuration of MuJava. For consistency with other research papers, we used only MuJava's method (code) level mutation operators, not its class level mutation operators. Furthermore, we reduced the number of equivalent mutants by disabling MuJava's AOIS mutation.

MuJava created 10,026 mutants from the 72 mutated classes. We evaluated these mutants against four test suites from 72 classes, resulting in over 2,887,488,000 test executions. Executing all of these tests on all of the mutants took a quad core server, executing continuously, over one month of calendar time.

E. Experimental Setup

To evaluate test suite size, we ran Randoop to generate three different test suites containing 500, 1000, and 2000 tests *per class*. We ran Randoop with an arbitrary but consistent random seed, so for a given class a smaller suite is a subset of a larger suite. Each suite includes some non-consistent tests (non-deterministic observations) that are disabled, as discussed in Section III-A. Thus, the number of tests that are run is about 1% smaller than the number of tests in the suite.

In these suites, test *case* size is conflated with test *suite* size: larger suites contain, on average, larger tests. The reason is that to create a new test, Randoop extends an existing test. Therefore, Randoop tends to add larger and larger test cases over time. To control for this factor, we created an additional test suite of size 500, by choosing the *last* 500 tests that were added to the 2000-test suite. It contains the same number of tests as the 500-test suite, but each test is larger on average. We call this suite the "last 500" suite.

F. Threats to Validity

Internal validity involves influences that can affect the results without the researcher's knowledge. Because of computational cost, the mutation analysis was only performed on the package selected by the Rata developers. Other packages might provide different results.

Threats to external validity are conditions that limit the generalization of the results. The primary threat here involves the use of only one industry program, which will not be

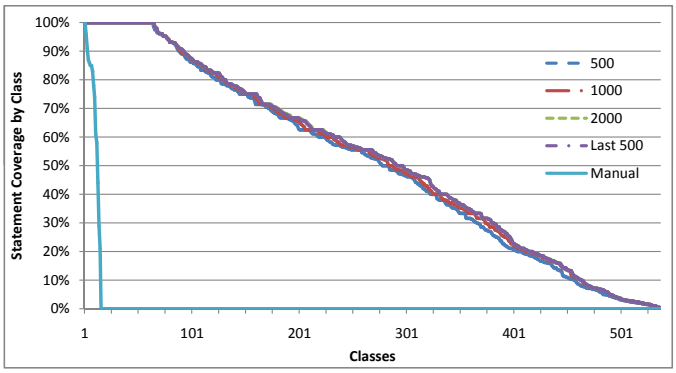

Fig. 2. Coverage is nearly the same for each of the generated test suites, and is much better than the manual suite, which covers only a few classes.

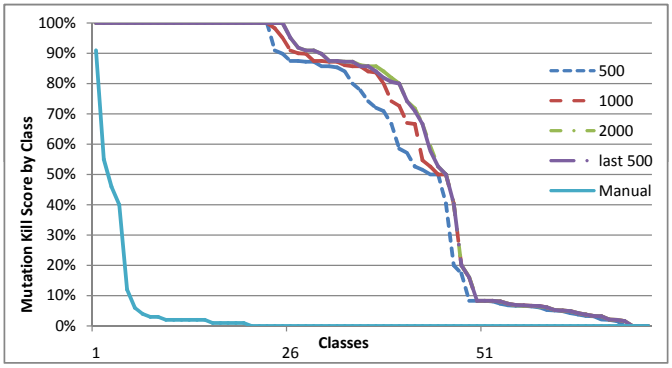

Fig. 3. Mutation coverage is similar for the generated test suites, and is much better than for the manual suite.

representative of all industrial programs. To address this threat, we included JHotDraw, an open source program, in the maintainability analysis. Future work will be needed to verify that the technique provides similar results on other industrial software systems.

Threats to construct validity arise when measurement instruments do not properly capture what they are intended to capture. We addressed this threat in a few ways. First, the end to end process of generating the tests, executing them, and calculating the results was automated, providing a consistent process for each of the dependent variables in the study. Second, we ran randomly-selected tests outside the automated process to verify the results were correct and consistent. Finally, some values and measures between separate test runs are not affected by the dependent variables, and we checked them to verify consistency between test runs.

V. Experimental Results

This section presents the results of the studies described in Section IV.

A. Test Suite Quality

1) Randoop for Non-library Programs: To address the first research question, we present the results of the best test suite that our enhanced Randoop tool generated. This suite was generated by Randoop with *string literals* and *removing lexically redundant tests* enabled, and *observer functions* disabled. The best test suite size, when considering test generation time, execution time, mutation scores, statement coverage, and maintainability, is the *last 500 tests* suite. This suite retains over 99% of the coverage and mutation scores of the full 2000 test suite, while executing more quickly and reporting fewer redundant errors when valid behavioral changes are made to the software over time.

This suite averages 52% statement coverage and 62% mutation kill score. Figures 2 and 3 show the coverage and mutation data at the class level, sorted from highest to lowest. These results answer the first research question by showing that Randoop can provide the Rata development team an automatically generated regression test suite that covers more than half the code and is able to detect a large number of potential defects.

The remaining subsections present the answers for the other research questions listed in Section IV-A. These answers also provide the rationale for the Randoop settings used in this section.

2) Effectiveness of String Literals: We calculated coverage and mutation scores for test suites generated with and without string literals support. For the vast majority of the classes in Rata, the impact of string literals on statement coverage was quite small. There are only six classes where the statement coverage was improved by more than 20%.

By contrast, string literals have a large impact on mutation kill scores. Across all of the classes of Rata, the mutation kill scores are approximately 10% better. Six classes had mutation scores go from 0% to 100% and five other classes improved by over 50%. On the other hand, two classes had a loss of around 10% for their mutation kill scores. These two classes only check strings for null, which Randoop now uses less frequently due to its larger pool of string values.

3) Effectiveness of Removing Lexically Redundant Tests: We created a test suite with Randoop configured to remove lexically redundant sequences before outputting a suite of the specified size, and compared it to a second set of tests created without removing redundant sequences. The difference in statement coverage between the two suites is quite small; only three classes had any difference. The mutation score, on the other hand, increases slightly for eight classes, while the remaining classes have no difference. These results make sense, as removing lexically redundant tests is supposed to reduce test suite size without materially affecting test effectiveness.

4) Effectiveness of Observers: To determine the effectiveness of specifying observer functions to Randoop, we randomly selected 10 classes from the mutated package, and manually determined which of their methods were observers. (For a description of observer methods, see Section III-D.) We then used Randoop to generate test suites of all four suite sizes, both with and without the observers specified.

Specifying observers to Randoop has very little impact on statement coverage, which is very high to start with. This makes sense, as the observers do not heavily impact test sequences, only the values Randoop uses for the test oracle. Observers have a larger impact on mutation score at smaller test suite sizes. The results show that the 500 test suite has a noticeable

difference in mutation score, while the 1000 test suite shows a smaller difference, and finally the 2000 and last 500 test suites show a negligible difference. These graphs are omitted due to space limitations.

Specifying observers to Randoop for small test suites can triple the number of assert statements each test contains. These additional *asserts* at the small test suite level are responsible for the increase in mutation score when observers are used.

In a larger suite, Randoop is more likely to have called all observers via its ordinary selection of random methods. There is little benefit from the explicit observer calls that Randoop adds when the user specifies observer methods.

5) Effectiveness of Test Suite Size and Test Case Length: This study involved evaluating the four different-sized Randoop test suites, as well as the manually written test suite, on Rata and computing the statement coverage and mutation scores. The results are shown in Figures 2 and 3.

Test suite size only slightly impacts statement coverage. For a given class, the average difference in statement coverage is only 3%. The last 500 tests provide the same statement coverage as the entire 2000 test suite. This shows that test case length is more important for statement coverage than test suite size.

The mutation kill scores are influenced more by test suite size. The scores increase the most between 500 and 1000 tests, and then the improvement decreases sharply for the 2000 test suite. For the majority of the classes, the last 500 suite has the same mutation score as the 2000 test suite, showing that test case length is also more important than test suite size for mutation kill score.

The generated tests perform significantly better than the manually-written tests for both coverage and mutation score. The Randoop test suites also execute significantly faster than the manual suite, since the manual suite is manually executed by clicking elements in the UI. Overall, the Randoop test suite will be a much better regression test suite for the Rata developers.

B. Test Suite Maintenance

In order to determine how much effort is required to maintain the generated test suites, we conducted experiments on Rata and on JHotDraw, an open source program commonly used in empirical studies in software engineering. We evaluated the maintainability of these tests in a few ways. First, we determined the effort required to maintain a Randoop test suite on both Rata and JHotDraw (Section V-B1). Next, we determined the impact that removing lexically redundant tests has on maintainability (Section V-B2) in Rata. Finally, we asked the Rata developers to examine the tests by hand to offer a qualitative analysis (Section V-B3).

1) Maintainability of the Regression Suite: For this evaluation, we ran the best Randoop regression suite (described in Section V-A1) on Rata version X and version $X+3$. Using the same parameters, we created a test suite for JHotDraw version 5.1 and ran it against version 5.2 and 5.3. Since the test suite is generated from a base version of the project, each subsequent version represents a new set of changes that need regression testing. For Rata, these two releases represent two years of development effort (and two intermediate releases). The Rata files we tested suffered 8.9% code churn: that is, 8.9% of the source lines were modified between the two releases. These changes included both bug fixes and new functionality. For JHotDraw, each version was released approximately one year apart. The code churn from 5.1 to 5.2 was 15%, and from 5.1 to 5.3 was 54%.

Program	Tests	Test failures	% tests failed	Defects	Behavior changes	False positives
Rata X+3	184619	6950	3.8%	1	2	0
JHotDraw 5.2	35966	467	1.3%	0	3	0
JHotDraw 5.3	40461	14564	36%	0	45	0

Fig. 4. Maintenance analysis. We ran Randoop on Rata version X and JHotDraw 5.1 to generate tests. The table reports the number of root causes for the test failures when running those tests on subsequent versions. In Rata version $X+3$ and in JHotDraw 5.2, there were only 3 root causes for all the failures.

There are three reasons a regression test may fail on a later version. (1) A regression defect was introduced during development. In this case, the developer must change the source code to fix the bug. (2) A desired behavior change causes the software to conform to a different specification, but the tests check for the old specification. In this case, the developer must change the test case to accommodate the new behavior. (3) The software is changed in such a way that it still satisfies the old specification, but the test was too strict and checked for the old implementation-specific behavior. This is a false positive, and the developer must generalize the test case to accommodate all permitted behaviors. Developers will resent any work required for case #3, and the work for case #2 should be kept to a minimum. The maintenance burden of a large number of failed tests caused by valid changes may lead developers to stop updating them, and they may abandon the regression suite over time.

Figure 4 shows the results of running the regression test suites on the later versions of the two programs. For Rata, our best Randoop test suite had 3.8% of tests failed. We believe this is a small enough number not to discourage a developer, especially because just a few edits can correct all the test failures. There are just three underlying causes for the failures found in version $X+3$. Furthermore, in practice regression tests would be run at least daily, not once every two years, so defects and behavioral changes would be noticed quickly when their impact is small and the code changes are fresh in the developers' minds.

Once they had investigated the failures, developers would regenerate the test suite from the new version of the code. This causes tests to be created for new functionality, while also automatically fixing tests that failed due to changed behavior.

For JHotDraw, version 5.2 had 1.3% failures, which is close to the results for Rata. Version 5.3 was different. In this case, there were 36% failures. JHotDraw 5.3 was a major change: its development site notes that projects created in previous versions may not work in the new version, and that they brought together

	Failures	% tests failed	Defects	Behavior changes	False positives
Redundant sequences	18000	4.1%	1	2	0
No redundant sequences	17113	3.9%	1	2	0

Fig. 5. The effect of lexically redundant tests on regression test failures.

three different development versions that forked off of 5.2. The vast majority of the failures were due to removed API functions that are easy to debug. Even with this extreme case, 64% of the tests still succeeded.

2) Maintainability of Removing Redundant Tests: For this evaluation, we ran Randoop both with and without elimination of redundant tests, as described in Section V-A3. We ran the resulting test suites on Rata version X and version $X+3$. The results are shown in Figure 5. When redundant tests are not eliminated, they are responsible for failure of 0.2% of the tests.

3) Human Assessment of Maintainability: We showed the Randoop-generated tests (from the 2000-test suite, without use of observers, and with lexically redundant tests) to Rata developers. While these developers are familiar with JUnit and unit testing and would like to have a unit test suite, as described in Section IV-C, there is no business case for writing one by hand.

Overall, the developers found the generated tests to be understandable, with acceptable style, code format, and coverage. The developers were surprised by the size of each test in the suite: the tests were longer than they expected. The developers may have expected a unit test to consist of a few lines of setup code followed by an assertion or two. While tutorial examples often look like this, much larger tests are required to test complicated behavior. Once they understood the behavior of the tests, the developers found the test length acceptable.

When we showed the developers the overall results from our experiments, they became very excited. They are now in the process of incorporating Randoop and its tests into their development process.

VI. Related Work

Our work builds on the feedback-directed random testing technique, and on the Randoop tool described in Section II. Feedback-directed random testing was first proposed and evaluated in the context of the Eclat [20] tool, which was itself inspired by the "operational abstraction" approach [12], [31] to measuring test quality. Eclat's main goal is to classify test results, in the absence of an oracle or specification, so that a human can be directed to examine the most promising ones. Eclat prunes sequences that appear to be illegal because they make the program behave differently than a set of correct training runs. Eclat's test generation differs from Randoop in that Eclat makes no assumption about a specification or oracle, but Randoop builds in known tests (contract checking), such as for the behavior of the `equals()` method. Randoop also has more heuristics for directing the random search, and has been ported to C# and extensively evaluated [22], [21], finding important errors in a variety of libraries. Our work differs in being focused on regression testing, maintainability, and real programs, domains for which Randoop was not previously suited.

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

Random testing Random testing [10] has been used to find errors in many applications; a partial list includes Unix utilities [15], Windows GUI applications [7], Haskell programs [1], and Java programs [2], [20], [18].

JCrasher [2] 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, so in practice it creates many invalid tests that throw an exception because of illegal input rather than because of a bug in the code under test.

Systematic testing Many techniques have been proposed to systematically explore method sequences [29], [3], [30], [9], [25], [4], [28]. Bounded exhaustive generation has been implemented in tools like Rostra [29] 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 permits a sequence to create some redundant and some nonredundant objects. 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. Randoop is scalable, but these tools are not: Rostra was evaluated on a set of 11 small programs (34–1000 LOC), and JPF's sequence generation techniques were evaluated on 4 data structures; neither Rostra nor JPF found errors in the tested programs.

An alternative to bounded exhaustive exploration is symbolic execution, implemented in tools like Symstra [30], XRT [9], JPF [27], DART [8], [25], 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. Some of these, like DART and jCUTE, even integrate random input generation into their symbolic execution approach, an idea investigated earlier by Ferguson and Korel [6]. 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.

Check-n-Crash [3] creates abstract constraints over inputs that cause exceptional behavior, and uses a constraint solver to derive concrete test inputs that exhibit the behavior. DSD [4] augments Check-n-Crash with a dynamic analysis to filter out illegal input parameters.

Comparing random and systematic Theoretical studies have shown that random testing is as effective as more systematic techniques such as partition testing [11], [17]. 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 [6]. Marinov et al. [14] 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. However, undirected random testing is a strawman; Randoop has been found to outperforms systematic techniques.

VII. Conclusion

We have presented an effective, fully automatic technique for creating a sensitive yet maintainable regression test suite. It is applicable to large, real-world software systems. It builds on the previously-known technique of feedback-directed random testing, and extends the Randoop implementation. It extends both the technique and the implementation to overcome a variety of limitations that we encountered while using Randoop. Our implementation is publicly available, under a permissive license, at http://randoop.googlecode.com/. Experiments with an industrial software system demonstrated that Randoop now creates test suites that have high code coverage and mutation kill score, that are comprehensible to developers, and that are easy to maintain even when years of changes are applied to the system under test. We also investigated the effects of various choices for Randoop's parameters, including test suite size, use of observer methods, and elimination of lexically redundant tests. We believe that the result is highly encouraging for the continued use of random testing, and feedback-directed random testing in particular, as one tool in the tester's toolbox.

References

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

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

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

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

[5] 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*, pages 59–68, Sep. 2006.

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

[7] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *USENIX Windows*, pages 59–68, Aug. 2000.

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

[9] 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*, Sep. 2005.

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

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

[12] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE*, pages 60–71, May 2003.

[13] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An automated class mutation system. *STVR*, 15(2):97–133, June 2005.

[14] 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, Sep. 2003.

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

[16] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *ESEC/FSE, New Ideas Track*, pages 496–499, Sep. 2011.

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

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

[19] C. Pacheco. *Directed Random Testing*. PhD thesis, MIT Dept. of EECS, June 2009.

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

[21] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *ISSTA*, pages 87–96, July 2008.

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, May 2007.

[23] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *ESEM*, Sep. 2011.

[24] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, July 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] B. H. Smith and L. Williams. An empirical evaluation of the MuJava mutation operators. In *TAICPART-MUTATION 2007*, pages 193–202, Sep. 2007.

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

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

[30] 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.

[31] T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *ASE*, 13(3):345–371, July 2006.