2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops

ArbitCheck: A Highly Automated Property-based Testing Tool for Java

Kohsuke Yatoh
The University of Tokyo
Tokyo, Japan
Email: k-yatoh@nii.ac.jp

Kazunori Sakamoto, Fuyuki Ishikawa
National Institute of Informatics
Tokyo, Japan
Email: {exkazuu, f-ishikawa}@nii.ac.jp

Shinichi Honiden
The University of Tokyo
National Institute of Informatics
Tokyo, Japan
Email: honiden@nii.ac.jp

***Abstract*—Lightweight property-based testing tools are becoming popular these days. With property-based testing, developers can test properties of the system under test against large varieties of randomly generated inputs without writing test cases. Despite the advantages of property-based testing, current property-based testing tools have a major drawback: they require developers to write generator functions for user-defined types. This is because it is difficult for a tool to infer the possible values for the type. However, user-defined generators sometimes fail to find faults by only producing overly limited varieties of values. In this paper, we present a new property-based testing tool, called ArbitCheck, which automates object generation by adapting the feedback-directed random test generation technique. With the help of feedback-directed random test generation, ArbitCheck exhaustively generates possible values of user-defined types and tests properties with them, so that it can reveal faults that are hard to find with either manually written tests or existing property-based testing tools.**

***Keywords*-Unit testing, Property-based testing, Random testing, Feedback-directed random test generation, Object-oriented, QuickCheck, Randoop, Java**

I. Introduction

Today, testing gains more and more importance as a method to ensure software quality. There are other methods to ensure quality, such as model-based verification or theorem proving, but most developers favor testing because it is simpler in concept and easier to deploy. The simplest way to test a program is to write test cases. Here, a test case for a program is a set of inputs to and expected outputs from the program. Popular testing frameworks including JUnit or RSpec support this kind of testing. However, these manually written tests have two disadvantages.

First, the test code tends to become very large when developers pursue quality seriously. Because one test case corresponds to one input, and there are many (and possibly infinite) inputs for a program, to gain confidence that the program works on any input developers have to write quite a lot of test cases.

Second, manually written tests cannot find faults caused by an input that is possible, but not expected by developers. From our experience, many faults are found when a program is given an input which developers failed to consider. Since test cases are written by developers themselves, in manually written tests there is no possibility that the program is tested against such inputs.

An alternative method to test software is property-based testing. In this method, developers write properties of a program that are expected to hold. Property-based testing tools generate a set of inputs and tests whether the properties actually hold against these generated inputs. The written properties are smaller than a set of test cases that describes the same properties, and it can hit unexpected inputs that reveal faults.

Lightweight property-based random testing [1], in which the properties are described by the implementation language itself rather than a modeling language, and inputs are generated at random, is originally developed for functional programming languages and becoming popular with the spread of functional programming languages. However, current tools require developers to provide generator functions to test user-defined types. Writing generator functions has several problems so that they may miss certain kinds of faults, as described in Section II-B.

In this paper we propose a property-based testing tool that automates the generation of user-defined types, called ArbitCheck. ArbitCheck solves object generation problems by employing feedback-directed random test generation [2] technique, which is an established method in the academic world. Also we addressed issues occurred when applying feedback-directed random test generation technique in practice. By using our tool, developers can test their software with less manual effort and enhanced fault-detection capability.

II. Background

A. Property-based Testing

Property-based testing is a method for testing, in which tests are described as properties of the SUT (system under test) that are expected to hold. QuickCheck [1] is the most famous tool for lightweight property-based testing. With QuickCheck, developers write properties as Haskell functions and test them with random values generated by the tool. QuickCheck had a large impact on the Haskell community, so that a standard Haskell textbook [3] devotes one chapter for testing with QuickCheck. It is also ported to many other programming lan-

978-0-7695-5194-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSTW.2014.68

guages, including Erlang[1], Scala[2], Java[3], Python[4], JavaScript[5], and so on.

List 1 is an example of properties written in Haskell from the original QuickCheck proposal [1]. Here, the `reverse` function takes a list as an argument and returns a reversed one. ++ is an operator that concatenates two lists.

```
prop_RevUnit x =
  reverse [x] == [x]
prop_RevApp xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
prop_RevRev xs =
  reverse (reverse xs) == xs
```

List 1. Example Properties of `reverse` Function That Should Hold

QuickCheck generates random lists, call the functions that express properties (i.e. `prop_RevUnit`, `prop_RevApp` and `prop_RevRev`), and checks whether the functions return true. If they return false, QuickCheck reports it to the user, because it indicates there are faults in the program or misunderstandings of properties.

Property-based testing has three advantages over manual testing. First, by writing only properties and omitting test data, developers can reduce the size of test code significantly. In a typical project using tests written manually, a series of test cases is required to check one software property and the total number of test cases becomes very large. In such a project, it is not uncommon that test code weighs over 50 percent of the overall code. Property-based testing remedies this situation by making test code simpler and shorter.

Second, by running property-based tests against randomly generated inputs, developers can find more bugs than manual testing. Property-based testing can check properties with lots of inputs so the chance of hitting bug is increased. In addition to that, property-based testing can find faults that are caused by corner cases which developers are unaware of. Manually written tests can hardly find this kind of faults, because the test cases are designed by the developers themselves. From our experience, most bugs are unexpected ones and caused when a program is fed an unexpected but valid input, so the ability to find these bugs offers a great help to developers.

Finally, the properties can be viewed as a lightweight formal specification of the program. Writing formal specifications promotes understandings of both specification and implementation, and is effective to share specification among the team. It is important that this specification is executable. This executable specification can avoid mismatch between specification and implementation, which is often the case if the specification is written in a separated document.

Despite these advantages, a problem arises when developers want to test user-defined types. It is not appropriate to generate completely random values for user-defined types, because user-defined types usually have restrictions on the possible values of their member fields.

[1]QuviQ QuickCheck http://www.quviq.com/index.html
[2]ScalaCheck http://www.scalacheck.org/
[3]junit-quickcheck https://github.com/pholser/junit-quickcheck
[4]pytest-quickcheck https://pypi.python.org/pypi/pytest-quickcheck/
[5]JSCheck http://www.jscheck.org/

```
data Frac = Frac Integer Integer -- numerator, denominator
fabs (Frac x y) = Frac (abs x) y
```

List 2. User-defined `Frac` Type and `fabs` Implementation

`Frac` type in List 2 is an example of such user-defined types. `fabs` is a function that returns the absolute value of the given `Frac`. `fabs` implicitly assumes that the numerator has the sign and the denominator is a non-zero positive value, so `fabs` does not return the correct value if the given `Frac` has a negative denominator. This is a fault to be fixed if the implicit assumption is not valid, but the implementation is correct if the assumption is actually valid. Since the validity of the assumption depends on the specification or the implementation of other parts, it is difficult for a tool to infer the correct behavior.

B. User-defined Generators

QuickCheck simply solves this problem by requiring developers to provide generator functions for user-defined types. These user-defined generators are expected to return only valid values for the type. For example, the generator for `Frac` may return only `Frac`s with non-zero positive denominators, but not `Frac`s with negative denominators. In this manner, QuickCheck can generate only valid values and filter out the possibility of false alerts caused by invalid values. However, user-defined generators sometimes have troubles when (1) the generator itself contains a bug, or (2) there is a gap between the valid values and the possible values for a type. Both leads to an unnecessarily limited distribution of the generated values, and inability to find faults that would be revealed by the values unnecessarily filtered out.

1) Generator Containing Bugs: It is not uncommon for a generator function to have a bug. This is because generating meaningful random values is more difficult than one expects. In fact, even the developer of a property-based testing tool may make a mistake in writing generators. Here is an example. ScalaCheck is a well-known and widely-used port of QuickCheck for Scala. ScalaCheck provides a default generator for Scala's `Option` type. However, the generator for `Option` had a long-hidden bug[6], which makes test results invalid. To avoid such issues, automated random test generation methods can be used to provide random but meaningful values.

2) Gap Between Valid Values and Possible Values: There are cases where the possible values for a type are not the same as the valid values for the type, peculiarly when the SUT contains faults. Here, we call a value *possible* if the value can be created within the normal execution of the SUT. List 3 is our motivating example. The `SumStack` contains two bugs, that cannot be found by property-based testing with user-defined generators.

`SumStack` maintains the summation of `data` in a cache field `sum`. So in valid states the summation of `data` must be equal to `sum`. Figure 1 illustrates some states of `SumStack`.

[6]https://github.com/rickynils/scalacheck/issues/75

```
public class SumStack {
  private List<Integer> data = new ArrayList<Integer>();
  private int sum = 0;
  public void push(int x) {
    data.add(x);
    sum += x;
  }
  public void pop() {
    // Bug 1. The developer forgot to update 'sum' in pop
    //sum -= data.get(data.size()-1);
    data.remove(data.size()-1);
  }
  public int getSum() {
    return sum;
  }
  public List<Integer> getData() {
    // Bug 2. The mutable reference to 'data' is returned,
    // so that the caller can modify 'data' freely.
    return data;
    // the correct implementation will be:
    //return Collections.unmodifiableList(data);
  }
}
```

List 3. Motivating Example: `SumStack`

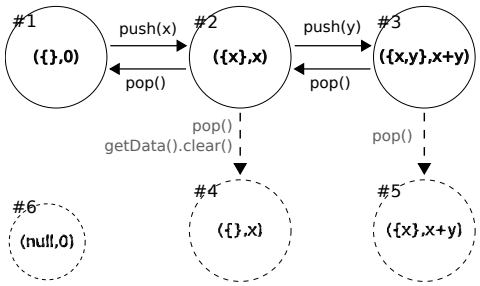

Figure 1. States of SumStack: States #1, #2, and #3 are valid and possible states. States #4, #5, and #6 are invalid. States #4 and #5 are possible only if the implementation of `SumStack` has bugs.

In each state ($\{a_i\}$,b), $\{a_i\}$ denotes the data in the stack, and b denotes the value of `sum`. The states with solid circles (#1, #2, and #3) are valid states, and the states with dotted circles (#4, #5, and #6) are invalid states. Possible values are all the states reachable from the initial state #1. Thus, states #4 and #5 are not possible if there is no bug but possible with the existence of bug 1 and 2. State #6 is invalid and not possible even with the existence of the bugs. Note that if `SumStack` have no bugs the invalid states must not be possible, because in terms of encapsulation `SumStack` is responsible to maintain consistency of its inner states. However, bugs 1 and 2 allow transitions to invalid states, making them possible values.

What should be the generator function for `SumStack`? List 4 is a simple intuitive generator that creates `SumStack` with a random length by calling `push` several times. Generator #1 can produce a variety of `SumStack` instances with valid states, and works perfectly if `SumStack` has no bug. Unfortunately, this generator is not effective if `SumStack` contains faults because instances generated by generator #1 can never reach states #4 and #5, so the property-based testing cannot reveal invalid transitions caused by `pop`. It is ironic that the generator aims to find bugs but does not work well if the SUT actually has bugs.

```
public static SumStack generator1(long seed) {
  Random random = new Random(seed);
  SumStack sumStack = new SumStack();
  int size = random.nextInt(100);
  for (int i = 0; i < size; i++) {
    sumStack.push(random.nextInt(100));
  }
  return sumStack;
}
```

List 4. Generator #1: Generator Only Using `push`

```
public static SumStack generator2(long seed) {
  Random random = new Random(seed);
  SumStack sumStack = new SumStack();
  int n = random.nextInt(100);
  for (int i = 0; i < n; i++) {
    switch (random.nextInt(2)) {
      case 0:
        sumStack.push(random.nextInt(100));
        break;
      case 1:
        if (!sumStack.getData().isEmpty()) sumStack.pop();
        break;
    }
  }
  return sumStack;
}
```

List 5. Generator #2: Generator Using Both `push` and `pop`

```
public static SumStack generator3(long seed) {
  Random random = new Random(seed);
  SumStack sumStack = new SumStack();
  int n = random.nextInt(100);
  for (int i = 0; i < n; i++) {
    switch (random.nextInt(3)) {
      case 0:
        sumStack.push(random.nextInt(100));
        break;
      case 1:
        if (!sumStack.getData().isEmpty()) sumStack.pop();
        break;
      case 2:
        sumStack.getData().clear();
        break;
    }
  }
  return sumStack;
}
```

List 6. Generator #3: Generator Using `getData().clear()`

The defect of generator #1 is that it missed transitions by `pop`. Generator #2 in List 5 tries to cover all transitions by calling `push` and `pop` in a random order. With generator #2, bug 1 can be found with property-based testing. However, bug 2 cannot be revealed, because the call to `getData().clear()` is not included in generator #2.

Generator #3 in List 6 tries to reveal bug 2 by adding calls to `getData().clear()`. However, writing generator #3 is in fact ridiculous, because the developer has to know that the call to `getData().clear()` is allowed and causes problems. In addition, generator #3 cannot be used when bug 2 is fixed, because `clear` on unmodifiable list throws an exception.

As we can see from the example of generators #1, #2, and #3, it is impossible to write a generator function that reveals both bugs 1 and 2 without knowing the existence of the bugs, and the generator that can reveal both the bugs cannot be used after the bugs are fixed.

C. Feedback-directed Random Test Generation

We solve this problem by automating the generation of user-defined types with the help of feedback-directed random test generation [2] technique. Feedback-directed random test generation explores the possible states of the type by combining calls to methods, so that it can find actual faults revealed by possible invalid states without reporting false alerts caused by impossible invalid states.

Feedback-directed random test generation is an algorithm to generate test cases. It takes a list of classes as an input, and output sequences of statements. A sequence corresponds with a test case, and consists of calls to the public constructors and methods from the class list, so that objects are created and mutated in each test cases.

It is notable that the created objects in each test case will be all possible, and must be valid in a correct program. This is because in object-oriented programming languages classes have to be encapsulated and any public method of the class must keep the object in a consistent state, or throw an exception if it is impossible to keep consistency. In other words, in a well-designed object-oriented program, if a sequence of public constructors and methods does not throw an exception, the resulting object must be in a consistent state. Also if the classes are well decoupled, all states must be reachable by using only public constructors and methods.

As the name indicates, feedback-directed random test generation monitors feedback from constructors or methods (i.e. exceptions thrown) to accelerate generation of sequences. If constructors and methods are randomly chosen without this feedback, it is likely that most sequences violate some object protocols and an exception is thrown during execution. Feedback-directed random test generation algorithm incrementally extends existing sequences, and stops extending if any exception is thrown, to efficiently create sequences that terminate normally.

Compared to systematic methods for generating test cases such as symbolic execution, feedback-directed random test generation is scalable with the size of SUT, because it treats a program as a black box. It is also easy to implement as it does not require any kind of static or dynamic analysis.

A Java implementation of feedback-directed random test generation algorithm is Randoop [4]. It also supports some optimizations to generate practical tests for real programs [5]. Randoop was originally designed to find faults with contracts, however, we extended Randoop to run property-based tests because property-based testing has the advantages described in Section II-A.

III. Proposal

We propose ArbitCheck[7], a property-based testing tool for programs written in Java language. It can test properties written in Java language without user-defined object generators. The key idea is to use feedback-directed random test generation algorithm for creating random inputs to property-based tests.

Here is an example of a property written in Java.

```
public class ReverseProp {
  @Check
  public static void prop_RevRev(int[] xs) {
    assertArrayEquals(xs, reverse(reverse(xs)));
  }
}
```

List 7. Example of Property Written in Java

The user compiles the code into a class file, and runs ArbitCheck with the class. ArbitCheck scans the class to find properties annotated with `@Check`, and then runs feedback-directed random test generation. The test generation ends when each property is tested against n different inputs, where n is a configurable integer number (defaults to 1000). After running property-based tests, ArbitCheck reports the result as follows.

```
prop_RevRev: OK, passed 1000 tests.
```

List 8. Report from ArbitCheck When Tests Passed

This indicates that `prop_RevRev` is checked with 1000 randomly generated inputs and all the tests passed. If some tests failed, ArbitCheck reports such that:

```
prop_RevRev: Failed 10 tests out of 1000 tests.
```

List 9. Report from ArbitCheck When Tests Failed

In this case, the failed 10 test cases are written to a JUnit test case file so that the user can inspect the failed inputs.

A. Writing Properties

The properties are written as a method with arbitrary signature. The property can take any number of arguments. In fact, the property is not necessarily static. Here is an example of a non-static property.

```
public class Wallet {
  private BigInteger money;
  ...
  @Check
  private void prop_Money() {
    assertNotNull(money);
    assertTrue(money.compareTo(BigInteger.ZERO) >= 0);
  }
}
```

List 10. Example of Non-static Property

The property `prop_Money` describes invariant of a private field `money`. Note that the property is declared private, so that no external user of `Wallet` can call the property method. ArbitCheck uses reflection to call the properties, and the resultant JUnit test cases do that, too.

If the property throws an exception, ArbitCheck regards it as an indication that the property failed to hold. In this manner, developers can use any assertion utility library, including JUnit, to describe properties. The exception can be either checked or unchecked, and the `throws` clause of properties does not affect any behavior of ArbitCheck.

A property can be declared that it does throw an exception as an expected behavior, and does not hold if no exception is thrown as in List 11.

[7]https://github.com/kohyatoh/arbitcheck

```
@Check(expected=EmptyStackException.class)
public static void prop_Overpop(Stack stack) {
  int size = stack.size();
  for (int i = 0; i < size; i++) stack.pop();
  // the (size+1)-th pop() should throw an exception
  stack.pop();
}
```

List 11. Example of Property Expecting Exception

ArbitCheck provides a way to write a conditional property by treating some kinds of exceptions specially. These special exceptions include ones thrown by `org.junit.Assume`, but can be configured by users. If these exceptions are thrown, the checking attempt is not accounted to the total number of inputs fed to the property.

```
@Check
public static void prop_PopSize(Stack stack) {
  assumeTrue(!stack.isEmpty());
  int size = stack.size();
  stack.pop();
  assertEquals(size-1, stack.size());
}
```

List 12. Example of Conditional Property

In this case, calls to `prop_PopSize` with an empty stack is ignored so that ArbitCheck runs until `prop_PopSize` is tested against 1000 non-empty stacks.

B. Monitoring Generated Objects

It is essential for random testing to monitor objects generated, in order to understand the distribution of generated objects. ArbitCheck provides a monitoring feature through test case labeling in the similar manner with QuickCheck. ArbitCheck offers `Monitoring` object, which has two methods, `collect(Object o)` and `classify(boolean b, String name)`. These methods add a label to the current test case. In order to use monitoring, properties must be marked that `monitoring=true` and take `Monitoring` object as the first argument.

```
@Check(monitoring=true)
public static void prop_RevRev(Monitoring m, int[] xs) {
  m.collect(xs.length);
  assertEquals(xs, reverse(reverse(xs)));
}
```

List 13. Example of Monitoring Using `collect`

By using `collect`, the user can make a histogram of objects generated. List 13 is an example property using `collect`, and the report from ArbitCheck will be:

```
prop_RevRev: OK, passed 1000 tests.
35% 0
25% 1
15% 2
...
```

List 14. Report from ArbitCheck When Using `collect`

To split test cases into some classes, `classify` is useful. `classify` adds a label to the test case conditionally.

```
@Check(monitoring=true)
public static void prop_RevRev(Monitoring m, int[] xs) {
  m.classify(xs.length == 0, "trivial");
  assertEquals(xs, reverse(reverse(xs)));
}
```

List 15. Example of Monitoring Using `classify`

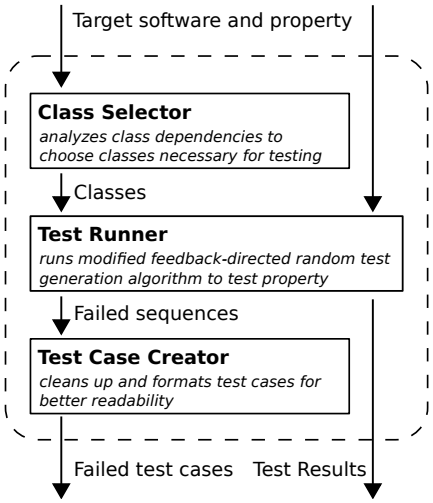

Figure 2. ArbitCheck Architecture

The report from ArbitCheck will be:

```
prop_RevRev: OK, passed 1000 tests (35% trivial).
```

List 16. Report from ArbitCheck When Using `classify`

A test case may have two or more labels at the same time. In this case, ArbitCheck uses the concatenation of labels in order to classify it.

IV. Implementation

Although the idea of ArbitCheck is very straightforward and the technologies behind it are established ones, there were some difficulties in implementing ArbitCheck. The difficulties mainly come from adapting feedback-directed random test generation algorithm to real world settings. In this chapter we explain the implementation of ArbitCheck.

ArbitCheck consists of three components, (1) class selector, (2) test runner, and (3) test case creator. Figure 2 illustrates the relationship between these three components. Class selector analyzes the SUT to make a list of classes that should be used to generate tests. Then the list is passed to test runner, in which feedback-directed random test generation is used to generate objects and check properties. Failed test cases are minimized and formatted as JUnit test cases by test case creator. Finally, results of the tests and failed test cases are reported to the user.

A. Class Selector

In practice, it is essential for random testing to appropriately designate the list of classes to use. If some necessary classes are not included, some possible objects cannot be generated, whereas unneeded classes in the list slow down the test generation. In addition, real applications contain codes that are not suited to random testing [6], for example, file system operations, thread manipulations, GUI-related classes or reflection mechanisms. By using `SecurityManager` of JVM we may partly sanitize these codes, but it is better not to

use them at all. Despite its importance, providing the class list is a laborious, tedious and error-prone task, thus it is necessary to automate the creation of this list.

Our goal is to call property functions with many different objects. If the parameter is a primitive, the tool can easily produce a random value. On the other hand, if the parameter is an object the tool has to create and mutate the object. In general, classes have dependencies, that is, objects with some other types are needed to create or mutate an object with a certain type. In order to analyze the dependencies, we define three terms, *provider*, *parameter*, and *mutator* for class *c* as follows:

$$provider(c) = \{c' \in C | \exists m \in member(c'), ret(m) = c\}$$
$$parameter(c) = \{c' \in C | \exists m \in member(c), c' \in param(m)\}$$
$$mutator(c) = \{c' \in C | \exists m \in member(c), ret(m) = c'\}$$

Here *C* denotes all classes in the SUT, *member(c)* denotes the methods and constructors of class *c*, *param(m)* denotes the parameter classes of member *m*, and *ret(m)* denotes the return type of member *m*. *provider(c)* is the set of classes that can create an instance of class *c*. *parameter(c)* is the set of classes that are needed to call members of class *c*. *mutator(c)* is the set of classes that are returned from members of class *c*. Thus, if we want to use class *c* in random testing, we need

- *provider(c)* to create instances of class *c*,
- *parameter(c)* to call methods or constructors of class *c*,
- and *mutator(c)* to mutate instances of class *c*.

We begin with the parameter of the property, and then add the *provider*, *parameter* and *mutator* of already added classes recursively until all necessary classes are included.

Here comes a complication. For *mutator* and *parameter*, we have no choice but include all of the classes in them. However, for *provider*, we do not have to include all of them. Actually we should not do so. For example, `Map` has a method `keySet()` that returns a `Set`, but creating `Map` is redundant to test `Set`. To choose appropriate classes, we introduce a rank-based heuristic approach. Rank for a member is defined as:

$$rank(m) = 1 + \sum_{t \in param(m)} rank(t),$$

and rank for a type is defined as:

$$rank(t) = \begin{cases} 0 & \text{(if } t \text{ is primitive)} \\ \min_{m \in M, ret(m)=t} rank(m) & \text{(otherwise)} \end{cases}$$

where *M* is the set of all members, that is $M = \{m | c \in C, m \in member(c)\}$. The rank for a type corresponds to the minimum number of method invocations required to create an instance of the type, and the rank for a method corresponds to the minimum number of method invocations required to invoke the method. These values can be computed efficiently using dynamic programming. When choosing classes from *provider*, we include only the classes with the lowest rank. This is because we assume that the lower rank indicates the lower level of dependence, and that classes with lower rank is less likely to introduce unnecessary dependencies.

```
GenerateSequencesAndRunTests(classes, property)
  nonErrorSeqs ← {}
  while not isFinished() do
    m(T_1, ..., T_k)
      ← randomPublicMemberOrProperty(classes, property)
    seqs, vals
      ← randomSeqsAndVals(nonErrorSeqs, T_1, ..., T_k)
    if m is a property then
      checkProperty(m, seqs, vals)
    else
      newSeq ← extend(m, seqs, vals)
      if execute(newSeq) does not throw an exception then
        nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
      end if
    end if
  end while
```

Figure 3. Rough Sketch of Feedback-directed Random Test Generation Algorithm Modified for Property-based Testing: Changes from the original algorithm are underlined.

B. Test Runner

Test runner takes the class list produced by class selector, and conducts feedback-directed random test generation. It also monitors the results of property tests. Figure 3 is a rough sketch of the feedback-directed random test generation algorithm modified for property testing.

isFinished() returns true if the number of non-ignored property checking attempts reaches the predefined threshold. *randomPublicMemberOrProperty()* chooses the property or one of public methods and constructors of the classes at random and returns the selected one. *randomSeqsAndVals()* chooses variables that fit to the parameter of the chosen method from the existing non-error sequences and returns them along with the sequences they are belonging to. Then, a new sequence is created by *extend()*. If the chosen method is a property, the newly generated sequence is executed under a monitored environment by *checkProperty()*. Otherwise, the newly generated sequence is executed, and added to the set of non-error sequences if it does not throw an exception.

C. Test Case Creator

The test cases generated by ArbitCheck are intended to be inspected by developers. Thus, the readability of the test cases does matter. Test case creator applies several techniques to make the test case cleaner and more natural.

1) Removing Redundant Method Calls: From the nature of random testing, the test cases often contain redundant method calls that do not affect the result of tests. Such redundant method calls are typically, but not limited to, calls for getter methods whose return values are not used in the latter sentences. Randoop exports a feature to remove these method calls in a greedy manner. For each sentence of the test case, Randoop removes the sentence, runs the test again and checks whether the result of the test case changes. If it does not, the sentence can be safely removed.

2) Sorting Test Cases: It is common in random testing that one fault causes lots of test cases to fail. In such situations, developers want to inspect the least complex test case first to roughly locate the fault. Then more complex test cases should be used to inspect the fault in detail. ArbitCheck sorts

TABLE I
NAMING RULES

Test case class	${name of class to test} + "Test"
Test case method	"test" + ${name of property} + id
General variable	${type name of variable}
Variable set by getter	${name of getter}

the resulting test cases according to the size of sequences corresponding to them, so that the least complex test case comes first and the most complex one comes last.

3) Naming Tests or Variables: Good names are essential for correct program understanding. We give the names based on the rule described in Table I. We name test cases based on the test target. Variables are named by its type. There is an exception for variables assigned by getter methods (i.e. methods with name "getXXX"). The "XXX" is often more descriptive than the type name so we use it instead.

V. RUNNING EXAMPLE

List 17 is our motivating example along with a property description. Please note that in this case the property is written as a private non-static method of the `SumStack` class itself, rather than in a separate class dedicated for test.

```
public class SumStack {
  private List<Integer> data = new ArrayList<Integer>();
  private int sum = 0;
  public void push(int x) {
    data.add(x);
    sum += x;
  }
  public void pop() {
    // Bug 1. The developer forgot to update 'sum' in pop
    //sum -= data.get(data.size()-1);
    data.remove(data.size()-1);
  }
  public int getSum() {
    return sum;
  }
  public List<Integer> getData() {
    // Bug 2. The mutable reference to 'data' is returned,
    // so that the caller can modify 'data' freely.
    return data;
    // the correct implementation will be:
    //return Collections.unmodifiableList(data);
  }
  @Check
  private void prop_SumData() {
    int _sum = 0;
    for (int x : data) _sum += x;
    assertEquals(_sum, sum);
  }
}
```

List 17. Motivating Example With Property

`prop_SumData` asserts that the summation of `data` is always equal to `sum`. However, when we test this property by ArbitCheck, ArbitCheck reports the property does not always hold.

List 18 is a failed test case reported by ArbitCheck.

```
void testProp_SumData() {
  SumStack sumStack = new SumStack();
  sumStack.push(100);
  sumStack.pop();
  checkProperty(SumStack.class, "prop_SumData", sumStack);
}
```

List 18. Failed Test Case #1

By inspecting this test case, developers will notice that they forgot to adjust sum in `pop`.

After fixing bug #1, developers can rerun ArbitCheck. Then it reports another failed test cases as follows:

```
void testProp_SumData() {
  SumStack sumStack = new SumStack();
  sumStack.push(100);
  List data = sumStack.getData();
  data.clear();
  checkProperty(SumStack.class, "prop_SumData", sumStack);
}
```

List 19. Failed Test Case #2

As we can see in the test case, the list returned by `getData()` can be modified outside the class and may become inconsistent with the sum. The problem here is that `getData()` returns a mutable reference to a field of the class (i.e. `data`), which leads to an inappropriate accessibility granted to the caller. The correct implementation should use `Collections.unmodifiableList` to prevent changes from outside of the class.

VI. DISCUSSION

A. Performance

Although systematic test generation approaches such as symbolic execution can achieve higher coverage than random testing, they suffer from scalability problems because they have to analyze the inner control structure of a program. Real-world applications are often too complex to be analyzed in a reasonable amount of time. Feedback-directed random testing treats a program as a black-box and ignores the inner control structure of the program. Thus the technique can be applied to large-scale programs. In fact, widely-used large libraries like Jakarta Commons Collections (with 61KLOC) can be tested using Randoop [2].

B. Generated Object Distribution

We must stress that the effectiveness of property-based tests strongly depends on the distribution of generated objects. Property-based testing becomes most effective if the distribution of generated objects is equal to the actual distribution. However, in general, it is difficult to completely simulate the actual distribution so some extent of the actual distribution may be left untested. The amount or tendency of such areas vary a lot depending on the characteristics of the SUT. Thus, monitoring generated objects are very important to assess the effectiveness of property-based tests.

If the monitored distribution is much smaller than the expected distribution, there are two possible causes: (1) faults in mutator, which limit the distribution inappropriately, or (2) limitation of random test generation.

If some mutators of a class have faults, the generated distribution becomes smaller. For example, if `Stack.push` has a bug and throws `NullPointerException` every time it is called, the stack can never grow up, so ArbitCheck can only generate empty stacks. However, ArbitCheck cannot know this is an erroneous distribution as property-based tests do not have the construction to describe the expected distribution.

The developer is responsible for judging the distribution is desirable, and find faults if it is not.

There are some cases ArbitCheck fails to generate instances for a certain class, even if the mutators do not have faults. An extreme example is `java.lang.Class`. `Class` must be obtained by `Class.forName(name)`, where `name` must be a valid class name loaded to JVM. It is impossible for ArbitCheck to infer such restrictions and provide appropriate values, thus ArbitCheck will almost never instantiate `Class` object. In this case, developers can write a helper factory method that returns random `Class` objects so that ArbitCheck can test them, although this partly spoils the principle of our tool that unexpected objects should be generated and tested.

VII. Related Work

There are many tools for test generation, ranging from research prototypes to commercial tools [6]–[11], which indicates the huge demand on automated testing.

Pex [7] is a test generation tool for .NET framework. It employs parameterized unit tests [12], which is similar to properties. However, the way of generating inputs are quite different from ours, as Pex uses dynamic symbolic execution to generate arguments for the parameterized unit tests. Dynamic symbolic execution is powerful in finding errors that seldom occur with random inputs, but takes more time to run and is not scalable. In addition, Pex requires user-defined generators, so it is facing the same difficulty of writing generators as the traditional property-based testing tools are.

Fuzz testing [13] was first introduced to test reliability of Unix utilities and has been used widely in practice. A fuzz testing tool SAGE developed by Microsoft Research [8] was intensively used to test Windows 7 [14]. Fuzz testing differs from property-based testing in that it tests not each methods but the whole program as a process. Fuzz testing generates inputs as text- or byte-streams of data, feeds them to the process and checks that the process behaves correctly (for example, does not crash). Fuzz testing is effective for testing the robustness of a program as a whole, while property-based testing is useful for testing each unit of a program not violating properties.

There are more heavy-weight testing tools, for example, Smartesting CertifyIt [9]. In such tools the properties are written by dedicated modeling languages such as UML or statecharts, so that developers can test more complex properties (e.g. temporal properties) which cannot be expressed by the implementation language. In a large and quality-sensitive project the cost of writing properties in dedicated modeling languages can be compensated by the increased ability to find faults. However, the cost is often too high for small teams[8].

Several approaches are proposed to improve the performance of random testing. In adaptive random testing [15], distances of objects are measured and the most distant untested object is selected to test. Swarm testing [16] limits the set of operations so that objects with extreme states can easily be created. RecGen [17] suggests a sequence of methods that access the same object field so that the methods in a sequence become more relevant to each other. Dynamic symbolic execution can be combined with random testing to achieve higher coverage [18]. We may adapt these techniques to improve the performance of ArbitCheck, however, we have to carefully evaluate them because they are developed for and evaluated with contract-based testing or regression test generation, not property-based testing.

[8]Small teams with less than 10 members are recommended by modern agile strategies for software development, including Scrum and Extreme Programming.

VIII. Conclusion

In this paper, we presented ArbitCheck, a property-based testing tool for Java. ArbitCheck takes properties written in Java and tests them with feedback-directed random test generation. Additionally, it produces JUnit test cases for developers to inspect failed properties with concrete sequences of method calls that lead to errors. Our tool can reveal faults that are hard to find by using traditional manually written tests or existing property-based testing tools.

References

[1] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *ICFP*, 2000, pp. 268–279.

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007, pp. 75–84.

[3] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real World Haskell*. O'Reilly Media, Nov. 2008.

[4] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *OOPSLA*, 2007, pp. 815–816.

[5] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *ASE*, 2011, pp. 23–32.

[6] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *ICST*, 2013, pp. 362–369.

[7] N. Tillmann and J. De Halleux, "Pex: White box test generation for .net," in *TAP*, 2008, pp. 134–153.

[8] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, 2008, pp. 151–166.

[9] B. Legeard and A. Bouzy, "Smartesting certifyit: Model-based testing for enterprise it," in *ICST*, 2013, pp. 391–397.

[10] M. Oriol and F. Ullah, "Yeti on the cloud," in *ICSTW*, 2010, pp. 434–437.

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *ISSTA*, 2007, pp. 84–94.

[12] N. Tillmann and W. Schulte, "Parameterized unit tests," in *ESEC/FSE*, 2005, pp. 253–262.

[13] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[14] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *ICSE*, 2013, pp. 122–131.

[15] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Jan. 2010.

[16] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *ISSTA*, 2012, pp. 78–88.

[17] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with mut-aware sequence recommendation," in *ASE*, 2010, pp. 293–296.

[18] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for c/c++ using concolic execution," in *ICSE*, 2013, pp. 132–141.