A Divergence-Oriented Approach to Adaptive Random Testing of Java Programs

Yu Lin, Xucheng Tang, Yuting Chen, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{*linyu1986, xchtang*}*@sjtu.edu.cn,* {*chenyt, zhao-jj*}*@cs.sjtu.edu.cn*

***Abstract*—Adaptive Random Testing (ART) is a testing technique which is based on an observation that a test input usually has the same potential as its neighbors in detection of a specific program defect. ART helps to improve the efficiency of random testing in that test inputs are selected evenly across the input spaces. However, the application of ART to object-oriented programs (e.g., C++ and Java) still faces a strong challenge in that the input spaces of object-oriented programs are usually high dimensional, and therefore an even distribution of test inputs in a space as such is difficult to achieve. In this paper, we propose a divergence-oriented approach to adaptive random testing of Java programs to address this challenge. The essential idea of this approach is to prepare for the tested program a pool of test inputs each of which is of significant difference from the others, and then to use the ART technique to select test inputs from the pool for the tested program. We also develop a tool called ARTGen to support this testing approach, and conduct experiment to test several popular open-source Java packages to assess the effectiveness of the approach. The experimental result shows that our approach can generate test cases with high quality.**

I. Introduction

Random testing, a testing technique which advocates the idea that test inputs are selected from the input domain randomly, provides the testers with strong support in producing sufficient test cases for a program without bias [13], [16], [22]. With the benefit of low cost and high tool supportability in adopting a relatively simple testing strategy, random testing can be easily applied in practice, but its effectiveness may be quite low [14], compared with those of systematic testing approaches (e.g., model-based testing), because all program paths or blocks of the objective program may not be covered by executing the program using the test inputs selected randomly.

Adaptive random testing (ART) [8], which was proposed as a relatively sophisticated form of random testing, has drawn great attentions in academia and industry. ART was firstly proposed to test software or portions of software dominated by numerical processing, and it was on the basis of an observation that a test input usually having the same potential as its neighbors in detection of a specific program defect. The fundamental principle of ART is to select test inputs evenly across the input spaces, because an even distribution of test inputs in the input space allows finding faults through fewer test cases than with purely random testing [8]. A typical implementation of ART is given in [8]: ART makes use of an executed set and an candidate set, where the executed set includes the test cases that have been executed but without revealing any failure, and the candidate set includes test cases that are randomly selected. The executed set is initially empty and the first test case is randomly chosen from the candidate set. The executed set is then incrementally updated with the element of the candidate set, which is farthest away from all executed test cases, until a failure is revealed.

ART helps to improve the efficiency of random testing and has been shown to reduce the number of tests required to reveal the first error by as much as 50% over purely random testing [8], without incurring much overhead. Meanwhile, the application of ART to object-oriented programs (e.g., C++ and Java) has drawn attentions for the possibility of pervasively using of ART in industry. Ciupa and Meyer [11] have proposed in an approach to adaptive random testing of programs in Eiffel, where the distance between two objects is calculated based on their types as well as their matching fields. However, applying adaptive random testing to test case generation still faces strong challenges. One of the main challenges is how to create appropriate candidate objects for ART, since the input spaces of the objective programs are usually high dimensional. The dimensions of an input space depend on not only the number of input objects but also their fields, and therefore a large scale routine usually has an extremely high dimensional input space which is unable to be depicted by using a geometrical graph. For this reason, an even distribution of test inputs in a space as such is not easy to achieve. Furthermore, the weights associated with attributes of objects are necessary for calculating the distances, while their values are usually not rigorous. Thus, it is necessary to generate divergent candidate test inputs for ART and use rigorous weights when calculating the distances so that test cases with higher quality can be generated.

In this paper, we propose a divergence-oriented approach to adaptive random testing of Java programs to address the above challenges. The essential idea of our approach is to prepare for the tested program a pool of test inputs, and then to use the ART technique to select a sequence of test inputs from the pool for the tested program. A test input

is composed of values of primitive types and/or objects of classes that are used as the parameters of the routine. An important characteristic of the pool is that each test input in it is of significant difference from the others, and thus more features of the program can be examined by running the test inputs in the pool than by running the test inputs with less divergence. Special objects (e.g. boundary values) are also contained in the pool. The first test input is randomly selected from the pool, and a test input can be selected next if it is farthest away from all the test inputs that have been used. In order to show the effectiveness of our approach, we have also implemented the approach as a tool called ARTGen, which provides the testers with support in generating JUnit [2] test cases for adaptive random testing of Java programs.

We make the following contributions in this work:

- We propose an approach creating the test pools containing divergent objects as well as boundary values of the input space, and then to perform the adaptive random testing of Java programs. The approach is expected to benefit the testers in automatically generating executable test cases for testing Java programs.
- To support our testing approach, we develop a prototype tool called ARTGen to support the testing approach.
- We conduct an experiment to test several popular open-source Java packages in order to assess the effectiveness of the approach. The experimental result shows that our divergence-oriented approach and ARTGen can provide the testers with support in detecting program defects in a cost-effective manner.

The rest of the paper is organized as follows. Section II gives some background information about Adaptive random testing approach. Section III shows an example of using the proposed approach. Section IV provides more details about the divergence-oriented approach to adaptive random testing. Section V evaluates the effectiveness and efficiency of the proposed approach by introducing an experiment which was conducted to test several popular open-source Java packages. Section VI discusses some related work, and Section VII gives conclusions and points out future research.

II. Background of ART

ART [8] is a technique for selection of test inputs. Its performance usually varies for different failure patterns, where a failure pattern is a common geographical feature of those failure-inducing inputs in the input domain [6]. Failure patterns can be divided into three categories: point, strip and block. An example with two dimension input domain is shown as Figure 1. In this example, suppose a method takes two integers x and y as input. The input domain of the method is $0 \leq x, y \leq 100$. Point pattern means that program will fail when x and y equal to particular integers, i.e., a specific point in the input domain, while strip pattern

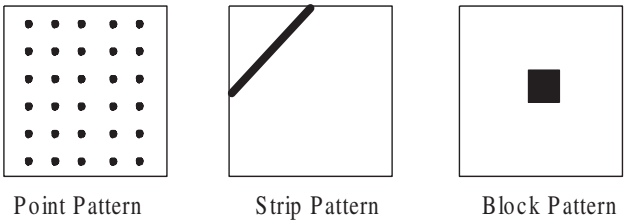

Figure 1. Three types of failure patterns with two dimension input domain.

may be of the form $10 \leq x + y \leq 20$ and block pattern may be of the form $30 \leq x, y \leq 50$.

ART is proposed on the basis of an intuition that using an evenly spread of test inputs for non-point types of failure patterns, random testing is more likely to detect failures. ART makes use of an executed set and an candidate set which are disjoint. The executed set is the set of distinct test inputs that have been executed but without revealing any failure; while the candidate set is a set of test inputs that are randomly selected. The executed set is initially empty and the first test input is randomly chosen from the input domain. The executed set is then incrementally updated with the selected element from the candidate set until a failure is revealed. From the candidate set, the element that is *farthest away* from all executed test cases, is selected as the next test case.

To distinguish which element is the farthest for a given element, ART algorithm should define the distance between two inputs. For numerical inputs, Euclidean distance can be used. For object inputs in an object-oriented program, object distance was proposed in [11]. For two objects in an object-oriented program, a distance can be defined by using objects' properties, including their types, fields, and the associated inheritance hierarchy. An object distance is a normalized weighted sum of the following three properties, for two objects o1 and o2:

- The distance between their types, based on the length of the path from one type to another in the inheritance graph, and the number of non-common fields.
- The distance between the immediate (non-reference) values of their matching fields, using a simple notion of distance for basic types (difference for integers, Levenshtein distance [19] for strings).
- For matching fields involving references to other objects, computing object distance recursively.

With the use of distance for both primitive values and objects, ART picks out the *farthest* input from all executed inputs for current use.

However, a typical ART algorithm does not explain the approach to create the candidate test set that contains objects. Creating suitable candidate inputs at random is still a challenge, since the input spaces of the objective programs are usually high dimensional and rigorous weights

```
public class Account {
    int balance; int min; History hist;
    public Account (int balance, int min, History hist)
      throws Exception {
        if(balance < min)
            throw new Exception();
        this.balance = balance;
        this.min = min;
        this.hist = hist;
    }
    public void deposit(int amount){
        hist = new History (balance, hist);
        balance = balance + amount;
    }
    public void withdraw(int amount){
        hist = new History (balance, hist);
        balance = balance - amount;
    }
    public void clearAccount(){
        balance = min; hist = null;
    }
}

public class History {
    int balance; History prec;
    public History (int balance, History prec){
        this.balance = balance; this.prec = prec;
    }
    public int getHistoryCount(){
        if(prec == null) return 1;
        else return prec.getHistoryCount()+1;
    }
}

public class BankSystem {
    public static boolean transfer(Account a, Account b,
        int amount) {
        if(a.hist.getHistoryCount() > 5){
            a.withdraw(amount); b.deposit(amount);
            return true;
        }
        return false;
    }
}
```

Figure 2. A sample Java program.

associated with attributes of objects are necessary for calculating the distances. Moreover, generating testing oracles and executable JUnit test cases automatically is attractive for random testing of Java programs. We will address the above challenges in the following parts of this paper.

III. Motivating Example

In this section, we demonstrate a contrived Java program to give an intuition of our test case generation approach. Details of the algorithm will be given in Section IV.

We use the example in Figure 2 to show how ARTGen works for some complicated routines. This code fragment designing for a simple bank system consists of three classes: `Account`, `History`, and `BankSystem`. Each account has its current balance and minimal balance and the current balance must be greater than the minimal balance. A series of histories are associated with an account. Notice that class `History` is self-referenced. `BankSystem` only does one operation: transfer some money from one account to another. In order to exhibit the advantages of our approach, we add a constrain to this operation, that is, only those accounts with more than five histories have the privilege to transfer money to other accounts.

Our approach automatically generates unit test cases at random for such Java classes. The approach produces test programs which are composed of JUnit test cases, and each test case consists of randomly chosen sequences of method calls for each class under test. Each generated test program can be executed to test the classes, and re-executed later on for regression test. As noticed in [18], because of complex dependencies between classes in object-oriented programs, it is usually not possible to test a method or a class in complete isolation. ARTGen thus is able to generate test cases for several classes in one time.

ARTGen takes a list of class names, a time limit, and a serials of contracts defined by user as input. When it is executed, it generates as many test cases as possible in the given time limit. After the execution, ARTGen outputs several Java files, as *ARTGen1.java*, *ARTGen2.java*, and so on. Each file is a JUnit test case and at most contains 300 test methods `test1()`, `test2()` ... `test300()`.

When applying ARTGen to this motivating example and testing `BankSystem.tranfer` method, for example, we define an oracle that requires an account's balance to be greater than its minimum balance, no matter before or after the transfer operation. We also require that the method should not throw exceptions. ARTGen generates more than 200 test cases in one second and reports 31 failures in which there are two different errors. We extract two shorter sequences of calls which result in each error and obtain the following results. To make the result clear, we also change the parameter values and add some comments for more readability.

Error 1. The first argument of the transfer operation is **null**, which causes a null pointer exception.

```
public void test4() throws Throwable {
    Account var1 = new Account(162, 127, null);
    var1.withdraw(8);
    try{
        boolean var2 = BankSystem.transfer(null, var1, 47);
    } catch(java.lang.NullPointerException e){
        System.out.println("In ARTGen1.java, test4" + e);
        fail("An exception:
            java.lang.NullPointerException");
    }
}
```

Error 2. The transfer operation will violate the oracle that the balance in an account must be greater than its minimum balance.

```
public void test13() throws Throwable {
    History var1 = new History(211, null);
    Account var2 = new Account(153, 120, var1);
    var2.deposit(149);
    var2.deposit(31);
    var2.deposit(173);
    var2.withdraw(240);
    var2.withdraw(86);
    Account var3 = new Account(271, 0, null);
    var3.deposit(190);
    boolean var4 = BankSystem.transfer(var2, var3, 95);
```

```
    // Regression assertion
    assertTrue(var4 == true);
    // Test contract, inferior to the minimum balance
    assertTrue(var2.balance > var2.min);
    assertTrue(var3.balance > var3.min);
}
```

In Error 1, the transfer operation will throw a null pointer exception. We use a `try-catch` block to catch this exception and use `fail` statement to indicate the method violate the oracle that it should not throw exceptions.

In Error 2, ARTGen inserts two assertions *assertTrue(var2.balance* $>$ *var2.min)* and *assertTrue(var3.balance* $>$ *var3.min)* to represent the contract we have defined. Since the transfer operation violates the oracle that the balance in the account *var2* is 85 after the transfer, which is less than its minimum balance 120, the test case in Error 2 is failed. We also generate assertion for regression test. This kind of assertion is generated based on the execution result of the tested method. In Error 2, the first assertion is added for regression test.

As a contrary, without using divergence-oriented approach, even if we adopt ART technique in selecting objects, we may not generate test cases with high quality. The result of such case is shown as following:

```
public void test() throws Throwable {
    Account var1 = new Account(153, 120, null);
    Account var2 = new Account(271, 0, null);
    boolean var3 = BankSystem.transfer(var2, var3, 95);

    // Regression assertion
    assertTrue(var3 == false);
    // Test contract, inferior to the minimum balance
    assertTrue(var1.balance > var1.min);
    assertTrue(var2.balance > var2.min);
}
```

This kind of test cases are not useful because the method under test `BankSystem.transfer` always returns **false**. Such test cases haven't generated objects with significant differences, and fail to provide excellent candidate objects for ART technique to select from. Thus, the test cases generated without divergence-oriented approach may result in low coverage or may not find errors.

IV. ART Test Case Generation Technique

A JUnit test case consists of a serials of test methods. A test method contains one tested method call, a sequence of constructor calls or method calls that set up the states of the input objects, and several assertions about the result of the tested method call [23]. To generate such JUnit test cases, ARTGen involves several steps:

- Selecting a tested method $m(T_1, ..., T_k)$ at random among the public methods of the tested classes.
- Generating inputs[1] for the tested method in order to form the candidate objects.

[1]A method's inputs include its receiver and arguments.

```
1: function ObjCreation(type, currentDeep)
2:     if isPrimitive(type) then
3:         return pseudoRandom(type) or
4:             specialValue(type)
5:     end if
6:     if currentDeep > createDeep then
7:         return defaultObj(type) or null
8:     end if
9:     constructor(T_1, ..., T_k) ← randomConstructor(type)
10:    for all T_i ∈ {T_1, ..., T_k} do
11:        inputs ←
12:          inputs∪ ObjCreation(T_i, currentDeep + 1)
13:    end for
14:    ⟨obj, seq⟩ ← invoke(constructor, inputs)
15:    diversify(obj, seq)
16:    return ⟨obj, seq⟩
17: end function
```

Figure 3. Object Creation Procedure

- Using ART technique to select a subset of these objects or primitive types from candidate objects as the selected method's receiver or arguments for actual testing.
- Running the tested method.
- Attaining the outcome and generating assertions based on oracles and the outcome.
- Logging relevant inputs and outputs and transforming them into test cases (e.g. executable JUnit test cases).

This section describes a randomized test case generation technique based on such process.

A. Creation of Objects

Testing a class means testing a number of method calls on instances of the class [21]. Constructing such a call requires both a receiver and objects for its arguments, if the method takes arguments. We first consider the issue of object generation[2].

ARTGen maintains a pool for each type referred by the tested classes. Each pool has a user-settable value *min* along with it which determines the minimum size of the pool. If the number of objects in the pool is smaller than this size, ARTGen will create some new objects and add them into the pool. The creating strategy involves pseudo-random choices of primitive values or constructors.

If ARTGen decides to create an object because there are no enough objects in the pool, it uses the algorithm shown in Figure 3. The algorithm takes four steps: **first**, it selects one of the constructors of a given class at purely random; **second**, it creates argument values for this constructor, if

[2]ARTGen views all the primitive type as boxed type (e.g., `int` as `Integer`, `double` as `Double`), so all the inputs can be viewed as objects.

needed. Noting that some of these arguments may also be objects, in which case the algorithm may need to create these objects recursively; **third**, it invokes the constructor with the selected arguments to create an instance. This procedure might cause a failure (e.g., an exception). In this case, the algorithm will discard these fail objects and go to step 1 to create another new object; **finally**, it diversifies the newly created object by calling its public methods[3].

In step 2, we use some guides to recursively choose arguments for constructors; we show details of this part in Section IV-A1 and Section IV-A2. In step 4, we diversify the newly created objects to attain divergent objects; this part is shown in Section IV-A3.

1) Generate arguments for constructor: The arguments of a constructor may be primitive types or reference types. When it is a primitive type, ARTGen chooses either a special value with a predefined probability, or a pseudo-random value. Special values are those values which are more likely to reveal a fault, e.g., $Integer.MAX\_VALUE$, 0, 1, and boundary values etc. Moreover, since different routines are used differently, the actual input domains of these primitive types may be different. Thus, it is unwise to generate all the primitive values in a uniform manner. For example, in Figure 2, the constructor of class `Account` takes an integer `balance` as argument. If the constructor uses an oracle that the argument `balance` must meet the precondition between 0 to 100, it is unnecessary to randomly generate `balance` based on uniform distribution in the integer's range. To adapt the various input domains of the primitive types, ARTGen provides some user-settable parameters to define the range of each field of primitive type, i.e. the boundary of an input domain. For example, in the constructor of `Account`, user can define the range of integer `balance` to be from 0 to 100. When the constructor needs a reference type, ARTGen recursively creates new object of this reference type. Like primitive type, we use **null** as a special value for reference type. Note that ARTGen does not select arguments of constructor from the pool, but creates purely new ones.

2) Recursive constructor: If a constructor needs a reference type as an argument, it may face a recursive creation of objects. For example, in Figure 2, the constructor of class `Account` needs an object of class `History` as an argument. And the constructor of class `History` also needs an object of class `History` as an argument. In this case, a recursive construct process may occur. To deal with this problem, we predefine a parameter *create-deep* to represent the depth of the recursion. In the example above, if we set *create-deep* to 4, the recursive depth will be 4, namely, a new instance of `Account` will at most maintain a `History` chain with the size of 3.

[3]We do not diversify the objects of primitive types, since the pseudo-random number generator can generate them uniformly.

3) Diversify the objects: Intuitively, more diverse objects can lead to higher coverage in random testing. Suppose we want to test method `BankSystem.transfer` in Figure 2. If all the objects of type `Account` in the pool only maintain a `History` chain with the size of 3 at most (i.e., the parameter *create-deep* is set to 4), the branch in the `if` statement will never be tested, since the bank system requires that only accounts with more than five histories have the privilege to transfer money to other accounts. Thus, diversifying the states of the newly created objects will be necessary to obtain higher coverage. Another important effect of diversification operation is that it can provide better choice for adaptive random test selection, which will be discussed in Section IV-B.

To obtain more diverse objects in the pool, ARTGen calls some public methods on an object. A public method is randomly chosen, and then called for 0 to *max-call* times, where *max-call* is a user-settable parameter. If this public method we have chosen needs arguments, we create them at purely random or use special values, as we create arguments for constructors. The whole diversification process will be repeated for 0 to *max-diversify* times, where *max-diversify* is also a user-settable parameter. For example, in Figure 2, if we set *max-call* = 4 and *max-diversify* = 3, ARTGen may repeat the diversification process for 2 times. At the first time, it may call method `deposit` for 3 times and at the second time, it may call method `withdraw` for 2 times. After this diversification operation, the newly created objects of type `Account` may cover the `if` branch when testing the method `BankSystem.transfer`. Notice that some public method of an object may not change its states such as the *getter* methods used to get the states of the object. Also, some public method may restore an object to its original state, e.g. the method `Account.clearAccount` in Figure 2. ARTGen provides a way to eliminate these unwanted public methods when proceeding diversification operation. Users can specify these unwanted public methods by annotations before they generate test cases. For example, if we mark the method `Account.clearAccount` with annotation `@discardMethod`, ARTGen will discard this method in the diversification operation.

If the method under test returns a value or an object, this value or object will be added into the corresponding object pool for later selection, because we think calling the tested method is also a diversification operation and its return value is possibly unique.

4) Construct String and Array: To construct an array, its length *n* will be randomly generated and then *n* objects with the corresponding types will be created recursively. `String` will be treated specially in order to improve efficiency. It will only be constructed by an array of characters, instead of choosing a constructor randomly, because we find the constructors of `String` have the same function.

B. Adaptive Random Test Inputs Selection

Many method calls require inputs including receivers and arguments. They can be either object references or primitive values. For both objects and primitive values, the strategy is to get an object or a primitive value from the pool using adaptive random testing strategy. ART spreads out the selected values over the corresponding intervals and selects better objects from the candidate objects for actual use. For example, integers should be evenly spread. Although notions of object distance to determine how far an object is from another are defined in [11] and used as a basis for object selection strategies, we have to adapt them for higher efficiency when using them for Java objects.

As illustrated in Section II, there are two kinds of attributes of an object, i.e. the type of the object and the fields in the object, that will influence its distance with others. The original notions are weighted averages using many weights to assess how each attribute of the object contributes to calculate distance. However, since there may be too many attributes (e.g. fields) in an object, it is inconvenient to set each of them by users before generating test cases. An easy approach is to set all these weights to one, as suggested in [11]. But this is not practical, because real objects are so complicated that some attributes provide more contributions than others in calculating distance. For example, a container's elements contribute much more than its size, so we need larger weight for elements. ARTGen uses annotations to set such weights. Based on the notions defined in [11], ARTGen provides two kinds of annotations: `typeWeight` and `fieldWeight`. The former is set on the class to assess the contribution of the type of the object, while the latter is set on the fields to determine the contribution of each field of the object. Thus, user can set different weights for different attributes of an object. All the default weight is 1, if we do not write any annotation. In Figure 2, if we mark the class `History` with annotation `@typeWeight(value=0.5)` and mark the field `History.prec` in class `History` with `@fieldWeight(value=10)`, it means the type weight of the objects of type `History` is set to 0.5 and the field weight of `History.prec` is 10.

We should also notice that a method may have preconditions, which are defined in test oracles, on its inputs. Let us name those inputs that meet the preconditions as *valid inputs*. In the following steps, we examine the inputs in the pool and only select *valid inputs*.

Using the distances and oracles, we select input in the following process. For each input in the tested method call, we keep a history list to record the objects that are already used. When we select an input for this method, we calculate the distances between each *valid input* in the corresponding pool and each object in the corresponding history list. Then, we sum up these distances and pick out the objects with the biggest sum as the current input. The selected object will be removed from the pool and added to the history list.

C. Writing Contracts as Oracles

Test oracles decide whether a test case has passed or failed. Devising oracles can be one of the most delicate and time-consuming aspects of testing [21]. Unfortunately, Java does not provide a direct way to write oracles. To simplify the definition and use of the oracles, we borrow the "contract" structure designed by Pacheco *et al.* in [25].

Contracts specify the expected effect of a method on the program states and they should be evaluated at runtime. The states include the precondition of a method which states the conditions to be established (by the callers) before any of its executions and the postcondition of a method which states conditions to be established (by this method) after execution.

To specify a contract, the user needs to declare a class implementing a `Contract` interface. The crucial method in this interface is `boolean checkPrecondition(Object[])` and `boolean checkPostcondition(Object[])` which returns `true` if the given objects satisfy the pre- and post- conditions of the contract. In ARTGen, the objects include the return value, the receiver, and the arguments of the tested method. User can indicate which methods a contract can apply to or implement a contract that can be applied to all the methods. Given a contract used for a tested method, ARTGen check preconditions before the execution and the postconditions after the execution to determine whether the current inputs are contract violative. For example, in Figure 2, if we want to create *valid inputs* of type `Account`, we must guarantee `balance` is not less than `min`. Thus, we can write contract as Contract 1 and add it to ARTGen when generate test cases.

Contract 1. The contract for the constructor of class `Account` which guarantee `Account.balance` $\geqslant$ `Account.min`.

```
public final class CheckAccount implements Contract {
    public Object checkPrecondition(Object[] objects) {
        Integer balance = (Integer) objects[0];
        Integer min = (Integer) objects[1];
        return balance >= min;
    }
    public Object checkPostcondition(Object[] objects) {
        ...
    }
}
```

D. Execution and Generating Assertion

After the inputs are selected, ARTGen will invoke the tested method and get the execution result. The tested method call may execute normally (including exit normally or throw an exception) or abnormally (e.g. executes longer than the predefined time limit because of infinite loop), and ARTGen only adds assertions for those normal executions.

There are two types of assertions provided by ARTGen. One is the assertions based on oracles. For example,

we have an oracle that `balance` must be greater than `min` in Figure 2. Thus ARTGen will generate assertion `assertTrue(a.balance > a.min)` where `a` is an instance of `Account`. The other is the assertions used in regression test. These assertions are generated based on the execution result of the tested method. If the method has a return value, we assert this value to be execution result; if the method throws an exception during its execution, we assert it should throw the same exception when applying the same inputs in regression test. Use method `BankSystem.transfer` in Figure 2 as an example. If its return value stored in variable `var` and the execution result is **false**, ARTGen will add assertion `assertTrue(var==`**`false`**`)` after the method call.

E. Generating Executable Code

To generate executable test cases, we record how a selected object is created. We use *creation sequence* to represent this creating process. The *creation sequence* records which constructor we have selected and recursively records how the constructor's arguments are created (the arguments' *creation sequence* precede the constructor call). The diversification operations are added orderly after the selected constructor. Once the *creation sequence* is constructed, we catenate the tested method call to the *creation sequence* and form a new sequence. Finally, we catenate all the assertions to the sequence. Thus, a sequence represents a test case and can be transformed to executable code.

F. Adaptive Random Test Case Generation Algorithm

We now summarize the overall process of our approach as a whole and present our algorithm for adaptive random test case generation.

The overall process is shown in Figure 4. The process starts from selecting a public tested method at random. Then, ARTGen creates objects and uses ART to select inputs. After that, ARTGen executes the tested method and attaches assertions. Finally, ARTGen outputs executable JUnit test cases.

Based on this process, the pseudo code of the algorithm is shown in Figure 5. We get a public method `method(`$T_1, ..., T_k$`)` from the tested classes at random at lines 3 and 4. Then, we select inputs using ART approach [11] for this method from line 5 to line 14. Note that if there is no enough objects in the pool $pool_{T_i}$, we call the procedure `ObjCreation` presented in Section IV-A to create some new objects at line 8. The argument *currentDeep* is set to 1 when it is the first time to call `ObjCreation`. After the inputs are selected, we invoke the method using these inputs and get execution results. If the method executes normally and its return value is not **void**, **null** or exceptions, we add it into the corresponding pool. This part is from line 15 to line 19. Since we attach the data structure *creation sequence* to each object, we can use it to generate executable code at line 20.

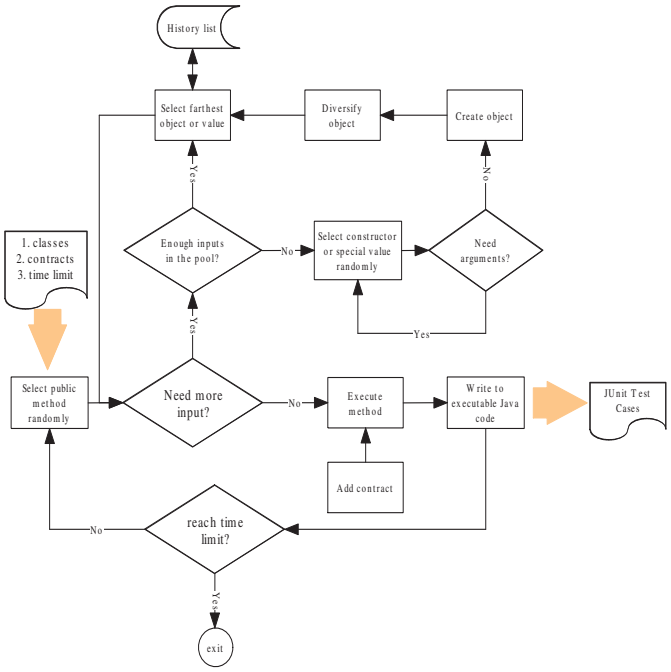

Figure 4. The overall process of adaptive random test case generation

V. Evaluation

To investigate the effectiveness of our approach, we have performed an experimental study on some widely used classes. All these classes are ranging from hundreds to thousands of lines of code.

A. Experimental setup

We use six subjects in the evaluation of ARTGen. Five of them are classes from Apache common library [1] which includes two subjects from package *Math* version 2.0, two from package *Lang* version 1.2, and one from package *Collections* version 3.2.1. The Apache common library is an Apache project focused on all aspects of reusable Java components and is publicly-available. The last subject is a small Java application named *Siena*.

Table I presents some properties of the classes under test. Column *'Package'* is the tested packages. The classes we test include all the classes directly in a package, but not include the classes in its sub-packages (e.g. when we test package *lang*, we don't test the classes in package *lang.text*, where *lang.text* is a sub-package in *lang*). Columns *'Loc'*, *'public classes'* and *'public methods'* represent total lines of code, number of classes, and number of public methods, respectively. Column *'description'* describes the use of the package.

To avoid writing contracts for these tested methods in the subjects, we use ARTGen's regression test feature to run our experiment. We first use ARTGen to generate test cases for all the original subjects. Then, we seed some errors manually

```
1: function GENTESTCASE(classes, contracts, timeLimit)
2:     while timeLimit not expired do
3:         method(T_1, ..., T_k)
4:             ← randomPublicMethod(classes)
5:         for all T_i ∈ {T_1, ..., T_k} do
6:             pool_{T_i} ← getPool(T_i)
7:             while | pool_{T_i} | <min do
8:                 pool_{T_i} ← pool_{T_i} ∪ ObjCreation(T_i, 1)
9:             end while
10:            input_i ←
11:                selectInput(pool_{T_i}, method, contracts)
12:            pool_{T_i} ← pool_{T_i} − {input}
13:            inputs ← inputs ∪ input_i
14:        end for
15:        ⟨obj, seq, normalExe⟩
16:            ← invoke(method, inputs, contracts)
17:        if normalExe = true then
18:            addToPool(⟨obj, seq⟩)
19:        end if
20:        writeToCode(seq)
21:    end while
22: end function
```

Figure 5. Adaptive Random Test Case Generation Algorithm

Table I
SUBJECT PROGRAMS

Package	Loc	public classes	public methods	description
Apache Commons				
math.geometry	340	5	46	3D calculation.
math.util	1161	7	48	Mathematic functions.
lang	4276	22	71	Basic utility.
lang.text	1475	5	152	Text processing.
collections.list	823	11	105	A container structure.
Java Applications				
Siena	1438	6	46	A wide-area event notification system.

to these subjects. Finally, we run these test cases generated for the original subjects on the versions with seeded errors. This experimental process only uses regression assertions we have shown in Section IV-D. If a regression assertion fails, we say the corresponding test case finds an error. We use uniform user-settable parameters in this example, e.g, pools' minimum size is set to 10, *create-deep* is set to 4, both *max-call* and *max-diversify* are set to 5. We use default values of `typeWeight` and `fieldWeight` except for containers. Since we think the elements in the container should contribute more than the size of the container when calculating distances, we set `fieldWeight` of the fields which denote the size of the container to 0.01. We also define the input domains for primitive values based on the feature of these subjects.

Table II
THE DIVERGENCE OF THE OBJECTS IN THE POOL

Class (package)	Average Distances ARTGen	Randoop	UnRand
Apache Commons			
ResizableDoubleArray (math)	5303	631	1029
Vector3D (math)	5982	1986	4877
PriorityBuffer (collections)	713	19	8
GrowthList (collections)	660	17	12
IntHashMap (lang)	3065	740	557
Overall averages	3145	679	1297

We measure two kinds of results. One is the divergence of the objects that ARTGen created. The other is the quality of generated test cases. The divergence is measured in terms of the distances of the objects in the pool: we measure the distance between each object pair in the pool (e.g. we will measure 45 pairs if there are 10 objects in the pool) and calculate the average distance. We select five representative classes in the above packages to measure the divergence. The quality of the generated JUnit test cases are evaluated according to seven factors: number of test cases generated in the same time limit, number of tests to find first fault, time used to find the first error, total run time of the test cases, totally found errors, errors reported by JUnit, and basic block coverage. We compare ARTGen with another two random test case generators: feedback-directed random test generator Randoop [25] and purely undirected random test generator (implemented in our ARTGen, and will be UnRand for brevity below), based on these seven factors.

All tests are run on a machine having a Pentium M 3.0 GHz processor, 1 GB of RAM, and running Windows XP SP2. Since the seed of the pseudo-random number generator influences the results, all the results presented below are averaged out over 5 2-minute tests of each subject using different seeds.

B. Result

1) Divergence of Objects: Table II shows the divergence of the objects created in the pool. Column *'Class'* is the classes we measure and column *'Average Distances'* is the average distances of those objects in the pool. From the result, we can find that ARTGen can outperform Randoop and UnRand in terms of the divergence of the objects in most cases. However, for class `Vector3D`, the objects generated by ARTGen and UnRand have similar distances. This is because `Vector3D` has a considerable number of constructors to create various objects. Therefore, the influence of the diversification operations provided by ARTGen is not remarkable.

2) Quality of Test Cases: Table III shows the comparison result on number of test cases generated in the same time limit, number of tests to find first fault, time used to find

Table III
RESULTS FOR THE SIX TESTED PACKAGE, BASED ON THREE APPROACHES, SHOWING THE NUMBER OF GENERATED TEST CASES, NUMBER OF TESTS TO FIND FIRST FAULT, TIME USED TO FIND THE FIRST ERROR, AND TOTAL RUN TIME OF THE TEST CASES

	Test case generated			Tests to first error			Time to first error (ms)			Total run time (ms)		
Package	ARTGen	Randoop	UnRand	ARTGen	Randoop	UnRand	ARTGen	Randoop	UnRand	ARTGen	Randoop	UnRand
Apache Commons												
math.geometry	1222	4948	6813	416	1079	3292	128	279	935	375	1279	1484
math.util	2391	766	9941	9	322	62	11	111	1160	3031	344	12672
lang	612	3198	1729	181	248	229	189	494	538	641	2493	1729
lang.text	1025	8487	3868	76	1182	849	42	734	411	391	3765	1909
collections.list	1828	5232	3455	142	1288	1080	57	582	440	312	1828	859
Java Applications												
Siena	904	6447	4478	27	674	343	81	610	471	985	4937	3235
Overall averages	1330	4846	4047	142	788	1246	85	468	659	957	2441	3648

Table IV
RESULTS FOR THE SIX TESTED PACKAGE, BASED ON THREE APPROACHES, SHOWING TOTALLY FOUND ERRORS, ERRORS REPORTED, BASIC BLOCK COVERAGE, AND TOTAL NUMBER OF ERRORS IN THE PROGRAMS

	Error-revealing test cases			JUnit reported test cases			Basic block coverage (%)			
Package	ARTGen	Randoop	UnRand	ARTGen	Randoop	UnRand	ARTGen	Randoop	UnRand	Total errors
Apache Commons										
math.geometry	2	1	1	13	58	31	79	75	84	2
math.util	4	1	4	19	3	123	83	70	87	4
lang	3	3	3	16	262	74	72	84	69	4
lang.text	3	3	1	37	115	112	87	82	72	3
collections.list	5	5	5	59	216	120	93	95	90	5
Java Applications										
Siena	2	1	2	82	416	130	80	69	83	2
Overall averages	3.2	2.3	2.7	37.7	65	98.3	83.5	79.2	84.2	3.3

the first error, total run time of the test cases. Columns '*Tests to first error*' and '*Time to first error*' indicate that adaptive random test generation can outperform undirected random test generation and feedback-directed random test case generation in terms of number of tests to find first fault and time used to find the first error. In most cases, ARTGen reduces the number of tests necessary to find the first fault by a considerable amount, and sometimes even by two orders of magnitude (e.g. package *math.util*). However, calculating the object distances and deciding which object is to be selected is time-consuming. This overhead causes that ARTGen generates fewer test cases over the same time, compared with Randoop and UnRand. Column *'Test case generated'* in Table III shows that in a fixed time, ARTGen generates much fewer test cases than Ranoop and UnRand. Due to this reason, it needs less time to run the test cases generated by ARTGen. Experimental data in column '*Total run time*' supports this finding.

Table IV shows the comparison result on total found errors, errors reported by JUnit, and basic block coverage. Column '*Total errors*' also records the total errors we seed into the code. Column '*Error-revealing test cases*' shows the errors each approach has found. We eliminate duplicated errors when counting these data. Column '*JUnit reported test cases*' shows the total errors reported by JUnit. From column '*Error-revealing test cases*', we can see ARTGen is more effective in finding errors: it can generate fewer test cases but find more or at least equal errors than Randoop and UnRand. In fact, column '*JUnit reported test cases*' indicates that Randoop and UnRand report much more errors, but most of these errors are duplicated. Test cases that find duplicated errors may not provide significant help in program unit test. Comparing the basic block coverage shown in column '*Basic block coverage*', we find that the results produced by three approaches are similar, and directed-random test (ARTGen and Randoop) attains a little higher coverage than undirected-random test (UnRand). Thus, in terms of basic block coverage, ARTGen is also more effective, since it attains nearly the same coverage through fewer test cases. Note that sometimes, Randoop could not generate enough test cases, e.g. for package *math.util*; or could not generate useful test cases, e.g. for package *Siena*, the coverage is much lower than the other two. We will discuss this problem later.

Figure 6 shows the information of Tables III and IV in histograms. The histograms compare the number of tests to find first fault (Figure 6(a)), time used to find the first error (Figure 6(b)), total run time of the test cases (Figure 6(c)), and basic block coverage (Figure 6(d)).

C. Discussion

We compare the quality of the JUnit test cases generated in the same duration by the three strategies. We have chosen several critical factors to measure the quality, including

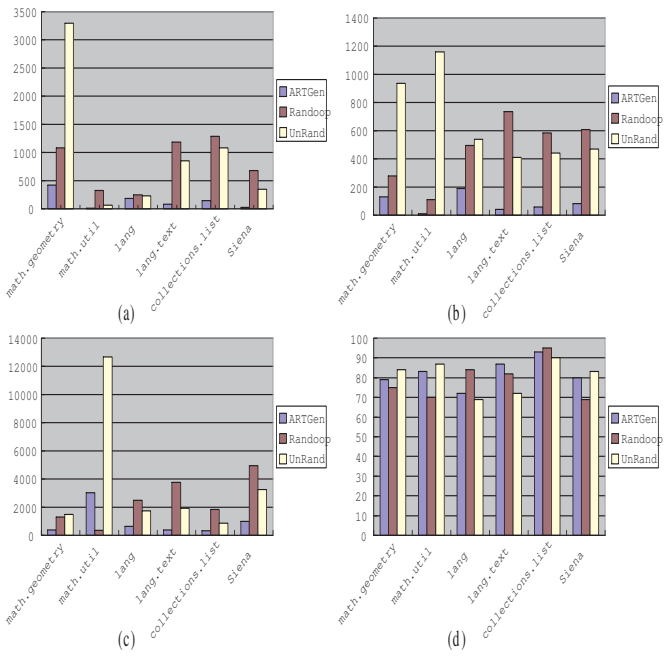

Figure 6. Comparison of (a) the number of tests to find first fault, (b) time used to find the first error, (c) total run time of the test cases, and (d) basic block coverage

the number of detected errors, run time and coverage. To measure the divergence of the created objects, we also calculate average distances of those objects in the pool. The results show that, comparing with feedback-directed random testing and undirected random testing, ARTGen can create more divergent objects, and with the help of ART technique, ARTGen generates fewer but higher quality test cases. These test cases can attain a similar coverage rate and find equal or more errors, while using fewer time.

The approaches of random test input generation can be divided into two main categories: top-down and bottom-up. A top-down approach selects a method and generates the input objects recursively, as described in Section IV-A. It is a goal-directed approach, since it guarantees that the objects needed for test inputs can be generated at the upmost possible. ARTGen belongs to this category. Bottom-up approaches use previously-generated objects as input objects (i.e. the return value of the tested methods) and store the modified objects to enable incrementally generation. The generator maintains a set of objects and their corresponding code sequences in a pool. To generate a test case, it chooses a method randomly and then selects test inputs from the pool. After the test case is generated, the new code sequence with the modified objects is added into the pool for reuse. Randoop, which uses feedback-directed random testing technique, is based on bottom-up generation.

A bottom-up generator is usually faster because it reuses previously-generated objects and reduces the time of trying to create valid input objects. As shown in Table III, Randoop usually generates more test cases than ARTGen and UnRand in the same duration. However, a top-down strategy will make the objects more diverse, because it generates each object from scratch. Table II shows that both ARTGen and UnRand create objects with larger distances than Randoop. Intuitively, diverse object can help to find faults faster. In contrast, the objects produced by a bottom-up generator are similar and may be the same sometimes, or the bottom-up strategy may fail to provide a non-null object because there is no such object in the pool. From Tables III and IV, we can see that Randoop cannot generate enough test cases for package *math.util* and also cannot attain a high coverage rate for package *Siena*. The reason is that most of the objects it generates are the same or similar and thus are not conducive to test.

D. Threats to Validity

Like any empirical evaluation, this study also has limitations which must be considered. Although we have experimented six packages and most of them are well known libraries, they are smaller than integrated Java software systems. Moreover, the errors used in our experiment are seeded manually. For this case, we cannot claim that these experimental results can be necessarily generalized to other programs with real bugs. On the other hand, threat of limited number of seeds may affect the accuracy of the experimental results. To reduce these kinds of threats, we have used 5 different seeds to generate pseudo-random numbers. The results presented here are obtained by averaging out over these 5 pseudo-random numbers.

VI. Related Work

In this section, we discuss some related work in areas of automatic test input selection and test case generation techniques.

Test input selection is often regarded as a critical step in random testing because selecting best input from candidate objects is helpful in eliminating bias. Some research [8]–[11] based on the idea of random testing tries to improve its performance by adding some guidance to the input selection. Adaptive Random Testing [8] and quasi-random testing [9] are two ideas that try to spread out the selected values over the input domain. Ciupa *et al.* [11] extend original ART algorithm which only solves numerical type selection to object-oriented software. Based on the ART intuition, a series of related algorithms have been proposed. Mirror ART [7] and lattice-based ART [20] reduce the overhead of ART. Restricted Random Testing [5] (RRT), which is based on restricting the regions of the input space where test cases can be generated, is also closely related to ART. As opposed to ART, where the elements of the candidate set are generated randomly, test cases are always generated so that they are outside of an exclusion zones (exclusion

zones are determined by the inputs in the executed set) in RRT. That is, a candidate is randomly generated, and if it is inside an exclusion zone, it is discarded and a new random generation is attempted. Chen develops a strong testing strategy that guides testers to select a set of test points so that the potential domain errors can be effectively detected [10]. With two rules for selection of test points evenly covering a specification domain, a tester is able to select test points that can effectively reveal program domain errors. However, all these selection techniques focus on how to select the best input from a pool contains enough candidate inputs, but do not consider how to create candidate inputs. We propose an divergence-approach to create candidate inputs which can be used by these selection techniques.

Test case generation techniques include systematic testing and random testing. For systematic testing, exhaustive exploration is the main problem and many techniques have been proposed [4], [26], [28]–[30] to solve this problem. Bounded exhaustive generation has been implemented in tools like Korat [4], Rostra [29] and JPF [28]. Rostra is a framework for detecting redundant unit tests by taking the use of the state of the receiver object and method arguments at the beginning of the invocation. Our ARTGen does not consider the redundant inputs because in most cases, the input domain is large and we don't reuse any objects except some special values. Thus, the redundance, if exists, will be few. Visser *et al.* also use state match to generate test input for Java containers, based on the Java PathFinder [27]. However, they do not use contracts to eliminate those illegal inputs as ARTGen does. Another approach for exhaustive exploration is symbolic execution which is implemented like Symstra [30] and jCute [26]. 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.

Random testing [17] can be more cost-effective than partition testing shown in some studies [13], [16] and has been used to find errors in object-oriented program [3], [12], [21], [24], [25]. RUTE-J [3] is a Java package providing tool support to programmers for randomized unit testing. In order to use RUTE-J, programmers write Java code in a form similar to JUnit test cases. Meyer *et al.* [21] implement AutoTest framework for Eiffel, a language realizes designing by contract. AutoTest uses Eiffel contracts as oracles and creates and selects inputs purely randomly. On the contrary, since Java does not provide the contract mechanism, ARTGen requires programmers to write oracles in Java code. These oracles are written in the form of Java classes which extend a particular interface. Meanwhile, ARTGen is not pure random, since it manipulates newly created objects and uses ART algorithm to select inputs. JCrasher [12] creates test inputs by using a parameter graph to find method calls whose return values can serve as input parameters. RANDOOP [25] uses a component set of previously-created sequences to find input parameters instead of a parameter graph. However, ARTGen does not reuse those objects which have been used as input. Instead, it creates new objects if there are no enough objects in the pool. But ARTGen may reuse the return value of the previous-executed methods (depending on the ART algorithm during the selection process) as RANDOOP. Eclat [24] can automatically produce oracles for a test input from the operational model which is provided by programmers. It reduces and classifies test inputs based on operational patterns. However, Eclat's performance is sensitive to the quality of the operational model, namely, a set of correct executions. ARTGen does not need any correct executions as input, but only some user-settable parameters.

Apart from the independent techniques of the systematic and random testing, some combinations of these two have been proposed. Godefroid *et al.* [15] explore a symbolic execution approach that integrates random input generation named DART. ARTGen is primarily a random input generator, but we can use techniques that impose some systematic rules in the input generation and selection to make it more effective.

VII. Conclusion and Future Work

In this paper, we proposed a divergence-oriented approach to adaptive random testing of Java programs. The essential idea of this approach is to prepare for the tested program a pool of test inputs, and then use the ART technique to select a sequence of test inputs from the pool for testing. In order to create objects with significant differences and provide suitable candidates for ART technique to select from, we apply some manipulations on the newly created objects. With the use of contracts and the execution sequences, we have also implemented the approach as a tool called ARTGen for JUnit, which provides the testers with support in generating test scripts for adaptive random testing of Java.

To demonstrate the effectiveness of our proposed approach, we performed an experiment on some well-known real software. The experimental result shows that ARTGen can create candidate objects with great differences and generate high quality test cases: it can find the first error with much fewer test cases and execute with less time than undirected random testing and feedback-directed random testing. Although ARTGen generates much fewer test cases in the same period, when comparing to the above two approaches, these test cases can achieve equal or even higher coverage.

In our future work, we consider to combine systematic techniques with random in object creation to get objects with more differences, and achieve higher coverage.

ACKNOWLEDGMENT

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 10 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058).

REFERENCES

[1] "Apache Commons," http://commons.apache.org/.

[2] "Junit, Testing Resources for Extreme Programming," http://www.junit.org/.

[3] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in *RT '06: Proceedings of the 1st international workshop on Random testing*. New York, NY, USA: ACM, 2006, pp. 36–45.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, vol. 27, no. 4. New York, NY, USA: ACM Press, July 2002, pp. 123–133.

[5] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *ECSQ '02: Proceedings of the 7th International Conference on Software Quality*. London, UK: Springer-Verlag, 2002, pp. 321–330.

[6] T. Y. Chen, D. H. Huang, and F. Kuo, "Proportional sampling strategy: Guidelines for software test practitioners," in *Information and Software Technology*, vol. 38, 1996, pp. 775–782.

[7] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," in *QSIC '03: Proceedings of the Third International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 4–11.

[8] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *ASIAN 2004: Proceedings of the 9th Asian Computing Science Conference*. IEEE Computer Society, 2004, pp. 320–329.

[9] T. Y. Chen and R. Merkel, "Quasi-random testing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 309–312.

[10] Y. Chen and S. Liu, "An approach to detecting domain errors using formal specification-based testing," in *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 276–283.

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: adaptive random testing for object-oriented software," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 71–80.

[12] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for Java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[13] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," in *IEEE Transactions on Software Engineering*, vol. 10, 1984, pp. 438–444.

[14] R. Ferguson and B. Korel, "The chaining approach for software test data generation." *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[15] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, vol. 40, no. 6. New York, NY, USA: ACM Press, June 2005, pp. 213–223.

[16] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence program testing," *Software Engineering, IEEE Transactions on*, vol. 16, no. 12, pp. 1402–1411, 1990.

[17] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[18] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck, and M.-H. Durand, "Testing levels for object-oriented software," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM, 2000, pp. 136–145.

[19] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, February 1966.

[20] J. Mayer, "Lattice-based adaptive random testing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 333–336.

[21] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, "Automatic testing of object-oriented software," in *SOFSEM '07: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 114–129.

[22] S. C. Ntafos, "On comparisons of random, partition, and proportional partition testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 949–960, 2001.

[23] C. Oriat, "Jartege: a tool for random generation of unit tests for Java classes," in *QoSA/SOQUA*, 2005, pp. 242–256.

[24] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *the 19th European Conference Object-Oriented Programming*, 2005, pp. 504–527.

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

[26] K. Sen and G. Agha, "Cute and jCute: Concolic unit testing and explicit path model-checking tools," in *CAV: Computer Aided Verification*, 2006, pp. 419–423.

[27] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 3–12.

[28] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for Java containers using state matching," in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006, pp. 37–48.

[29] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 196–205.

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