Patterns to automate tests for non-deterministic algorithms

ANDRÉ A. S. IVO, Centro Nacional de Monitoramento e Alertas de Desastres Naturais (CEMADEN), São José dos Campos, SP, Brazil
EDUARDO M. GUERRA, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil

Non-deterministic algorithms can return multiple distinct and valid results for subsequent invocations with the same input. These characteristics of non-deterministic behavior make the test automation for these algorithms quite complex. This paper presents a set of patterns that point to a viable solution to automate testing for systems with this type of behavior. This approach is structured using a set of 3 software patterns: Deterministic Characteristic Assertion pattern, that aims to create an assertion that verifies the validity of the algorithm result; Re-test With Different Seeds pattern that aims to execute the test several times with different seeds; Recycle Failed Seeds pattern, which introduces the persistence of the seeds used in failed tests to be reused in future test executions.

Categories and Subject Descriptors: Software and its engineering [**Software creation and management**]: Designing software—*Software design engineering*; Software and its engineering [**Software creation and management**]: Software verification and validation—*Software defect analysis*

General Terms: Development of Non-deterministic Algorithms

Additional Key Words and Phrases: development, non-deterministic, algorithms, patterns

ACM Reference Format:

André A. S. Ivo and Eduardo M. Guerra 2017. Patterns to automate tests for non-deterministic algorithms jn 2, 3, Article 1 (October 2017), 12 pages.

1. INTRODUCTION

When the execution of a software system should always lead to an expected result, it can be considered deterministic. In this case, from a known input, the algorithm must return at an expected output. However, in computer science these exists several algorithms whose execution can result in several valid outputs starting with the same input. The output and paths along the way, which depend on what these algorithms are called non-deterministic algorithms [Floyd 1967] and usually depend on various random decisions made during its execution.

Non-deterministic algorithms are used to solve problems that have several correct answers, and furthermore, can arrive by multiple paths to these responses. Some classic examples of non-deterministic algorithms are *primality testing*, *traveling salesperson*, *k-clique* and *multiprocessor scheduling* - all examples of NP-problems. This set contains solutions to basic problems in:

This work is supported by CNPq (grant 445562/2014-5) and FAPESP (grant 2014/16236-6)
Author's address: André A. S. Ivo, Centro Nacional de Monitoramento e Alertas de Desastres Naturais (CEMADEN), São José dos Campos, SP, Brazil; email: andre.ivo@cemaden.gov.br; Eduardo M. Guerra, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil; email: eduardo.guerra@inpe.br

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 24th Conference on Pattern Languages of Programs (PLoP). PLoP'17, OCTOBER 22-25, Vancouver, Canada. Copyright 2017 is held by the author(s). HILLSIDE 978-1-941652-06-0

—Graph theory

—Optimization

—Number theory

—Generic algorithms

—Algebra

Non-deterministic algorithms use functions that return a random number in a given range or a random element in a given set. In some cases, when this function is used only a few times, the use of automated testing becomes feasible, because it is possible to execute the algorithm simulating several return values for the random functions. However, if the random function is used several times or a variable number of times, it is not viable to simulate the value of several invocations in order to lead the execution to a given answer. Therefore, it is not practical to simulate all of these paths to test a non-deterministic algorithm with several random choices. Moreover,even if it were feasible to simulate all these answers, they would not be truly random. As a consequence, any development technique that rely on automated testing, such as Test Driven Development - TDD, is not viable to use in the development of such algorithms.

As any code with a lack of tests, when these algorithms need maintenance or new solutions need to be developed, faults can be inserted into the system without notice. In order to enable the detection of failures, the objective of this paper is to present a set of patterns that assist in the test automation of these non-deterministic algorithms.

The goal of this section is to present the proposed approach to automate tests of randomized logic. This approach is structured using a set of 3 software patterns. Section 3 presents the Deterministic Characteristic Assertion pattern, that aims to create an assertion that verifies the validity of the algorithm result. Section **??** the Re-test With Different Seeds pattern that aims to execute the test several times with different seeds. Finally, Section 5 shows the Recycle Failed Seeds pattern, which introduces the persistence of the seeds used in failed tests to be reused in future test executions.

The target audience for this paper are software developers and computer scientists who creates software with non-deterministic algorithms, such as genetic algorithms for optimization problems, concerned with code quality and test automation.

2. MOTIVATING EXAMPLE

As a running example that is used through the patterns, to represent a method whose return is non-deterministic, consider an algorithm that generates an array of “n” positions, with pseudo-random numbers varying between 10 and -10, whose total sum of elements is always zero. Note that in this example a number of possible returns grows exponentially with “n”. The following formula calculates the number of possible returns, where “n” is the array size and “e” is the number of elements:

$$Possibilities = e^n \tag{1}$$

Since between -10 and 10 we have 21 elements including 0, the following calculates the number of possibilities for an array of 3 positions:

$$Possibilities = 21^3 = 9261 \tag{2}$$

Each of the possibilities present a path that must be tested, so the amount of testing rises exponentially with the number of possibilities. In fact here the question is not the number of possible answers, but the fact that there are several correct answers and the algorithm can return different answers to each execution.

In the next section, more details will be presented regarding the set of patterns that document solutions for non-deterministic algorithm testing. Also, section 3 present the Deterministic Characteristic Assertion pattern, that aims at validating the behavior of the algorithm; section 4 presents the Re-test With Different Seeds pattern, that increases the test code coverage, and the section 5 presents the Recycle Failed Seeds pattern, that assists in regression testing.

3. DETERMINISTIC CHARACTERISTIC ASSERTION

3.1 CONTEXT

In non-deterministic algorithms the output result is different for each execution. The correct solution to the problem is determined by a set of characteristics that allow for checking the validity of the result. At each execution these algorithms make random choices, searching for a path that will deliver a valid result, based on the sets of characteristics.

3.2 PROBLEM

How to perform verification of characteristics that do not change, to ensure that the output is valid?

3.3 FORCES

—The result of the same execution may be different for the same entries, which make it difficult to compare with the return value;

—Obtaining a valid return for a test case does not mean a valid return in the following executions;

—There may be several varied behaviors and the amount of random decisions can also not be determined, making it impossible to create Mock Objects [Mackinnon et al. 2000; Freeman et al. 2004] that return fixed results for these decisions;

3.4 SOLUTION

Define a group of characteristics that guarantee the result is valid, and create an assertion test.

The Deterministic Characteristic Assertion pattern suggests that a set of criteria be defined for the tests, to evaluate the characteristics at the end of the execution. If all criteria are met, the results should be valid. This pattern can be associated with test oracles. An ideal test oracles provides an pass/fail judgment for any possible program execution, judged against a specification of intended behavior [Baresi and Young 2001].

This pattern represents the fundamental part of the general solution, and is related to the correctness of an algorithm. In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. Functional correctness refers to the input-output behaviour of the algorithm (i.e., for each input it produces the expected output) [Dunlop and Basili 1982].

The solution to the problem is simple, it is summarized in the selection of fundamental characteristics that determine the validity of the result of the algorithm.

An analysis of the algorithm must be performed with the intention of identifying the main requirements and/or behaviors that do not change, and that would guarantee a valid result. Then these items should be related to a list of fixed algorithm characteristics.

For each of the related items on the list, one or more correctness tests should be developed. After developing the set of characteristics tests, a pseudo-random input should be applied to the algorithm and submitted to the test set. If all set of characteristics tests run successfully, it means the output is valid, regardless of the value.

In general, there is intention of evaluating the behavior of the algorithm independently of the generated result. In some cases, it is possible to extract the characteristics of the result itself, in others, it is not. As a good practice, there may be cases where it is necessary to divide the algorithm into some parts so that it is possible to extract the fixed characteristics and then test them.

Figure 1 demonstrates how to use the pattern. It is possible to verify that the *testFixedCharacteristics* has been divided into some parts. In this case, the characteristics are obtained through the outputs of the intermediate functions. The characteristics can be identified as business rules and or functional requirements of the system.

The function *assertValueInterval()* and *assertSumEqualZero()* represents the point where all assertions of the characteristics are performed.

```
private void generateArrayBasedRandomSeedWithSumZero(Random seed, int input){
  int[] values = generateNumberInterval(seed, input);
  int result = sumEqualsZero(values);
}

private void testFixedCharacteristics(){
  int input = 10;
  //A pseudo-random input is generated based on failed seed
  Random rSeed = new Random(System.currentTimeMillis());
  int[] values = generateNumberInterval(rSeed, input);
  assertValueInterval(values, n);

  int result = sumEqualsZero(values);
  assertSumEqualZero(result, n);
}
```

Fig. 1. Deterministic Characteristic Assertion Algorithm.

Based on the motivational example, Figure 2 demonstrates the use of the pattern.

```
private void assertValueInterval(int[] arr, int arraySize) {
  int result = 0;
  //verify if value are between -10 and 10
  for (int i = 0; i < arraySize; i++) {
    assertTrue(arr[i] >= -10 && arr[i] <= 10);
  }
}

private void assertSumEqualZero(int[] arr, int arraySize) {
  int result = 0;
  //verify if sum equals 0
  for (int i = 0; i < arraySize; i++) {
    result = result + arr[i];
  }

  //verify the sum
  assertEquals(0, result);
}
```

Fig. 2. Motivation example using a Deterministic Characteristic Assertion Algorithm.

The assertion functions are simple and could be implemented in a single function. However, for didactic purposes it was divided into assertValueInterval and assertSumEqualZero, one to check if the generated numbers are between -10 and 10 and the other to check if the sum is equal to zero. Figure 3 shows the application of the characteristics assertion in the test function.

```
@Test
private void testFixedCharacteristics(){
    int n = 2;
    //A pseudo-random input is generated based on failed seed
    Random rSeed = new Random(System.currentTimeMillis());
    int[] result = ArrayFactory.generateArrayBasedRandomSeedWithSumZero(rSeed, n);

    assertValueInterval(result, n);
    assertSumEqualZero(result, n);
}
```

Fig. 3. Motivation example test.

3.5 CONSEQUENCES

—Checking the features is weaker than checking every possible value that the algorithm can take;

—Because of its random behavior, the fact it may be true for a certain execution does not mean it will be for all of them;

—It is possible to verify if an answer is valid even if you do not know exactly what the value will be;

—It may be necessary to divide the main algorithm to facilitate the extraction of characteristics.

4. RE-TEST WITH DIFFERENT SEEDS

4.1 CONTEXT

The Deterministic Characteristic Assertion pattern is related to the correctness of an algorithm, however, the test coverage is not solved by this pattern because of the difficulty of covering all possible inputs of the domain.

As mentioned, traditional testing strategies often focus on exploring the input domain, but unfortunately, full coverage of the input domain is not an option in our problem, because an infinite input field would lead to endless test cases, which is impractical.

On the other hand, there are techniques such as random testing, that are quick and applicable in an infinite domain, but do not necessarily explore the entire domain [Elva 2013]. Random testing involves choosing an input combination at random and testing the algorithm in sequence, until an error occurs [Hamlet 2002]. Even though it appears to be a naive strategy, random testing has been used for decades with good results in different contexts, and in some cases it is the only possible strategy anyway [Chen et al. 2010].

4.2 PROBLEM

How to increase the coverage of code in testing non-deterministic algorithms?

4.3 FORCES

—Because of non-deterministic behavioral characteristics, these algorithms have low test code coverage.

—With the high number of random decisions in the algorithm, the number of tests to perform is too high.

4.4 Solution

Write the tests using the Deterministic Characteristic Assertion pattern and repeat them with pseudo-random inputs, so that the algorithm is induced to go through different paths, increasing the test code coverage.

This pattern is focused on increasing code coverage, which is not the same thing as functional correctness, which is covered by the Deterministic Characteristic Assertion pattern. Code coverage is not an aspect of product quality, it is just one measure of how well exercised a product is, so better code coverage has a weak correlation with fewer expected undiscovered product bugs.

For this reason, this pattern only works when combined with the Deterministic Characteristic Assertion pattern.

To use this pattern, simply create a test using the Deterministic Characteristic Assertion pattern with pseudo-random inputs, and place it inside a loop so that it is repeated, as can be observed in Figure 6.

It is recommended to use controlled generation of pseudo-random inputs, and the way to do this is to introduce new seeds. In general, this functionality is present in almost all programming languages. A seed is generally an integer and acts as a “key-primary” in pseudo-random generations. When the pseudo-random generation algorithm receives a seed, following generations must always result in the same sequence, therefore the so-called “pseudo-random” generation.

The combination of pseudo-random inputs and the number of repetitions causes the algorithm to be exercised to traverse different paths, so the larger number of repetitions, the greater the test code coverage.

In Cases where the complexity of the code and the number of characteristics is very high, it is recommended to increase the number of repetitions in order to generate a better code coverage.

```
@Test
private void testFixedCharacteristics(){
  int n = 2;
  //First set of the input domain.
  //Number of repetitions.
  int numberRepetitions = 100;
  for(int i=1;i<=numberRepetitions;i++){
    //New seed is introduced based on the computer clock.
    long new_seed = System.currentTimeMillis();
    Random rSeed = new Random(new_seed);
    int[] result = ArrayFactory.generateArrayBasedRandomSeedWithSumZero(rSeed, n);

    assertValueInterval(result, n);
    assertSumEqualZero(result, n);
  }
}
```

Fig. 4. Re-test With Different Seeds Algorithm.

4.5 Consequences

—With greater number of repetitions the test code coverage is increased;

—With greater number of repetitions, the time of test executions is increased;

—It does not guarantee 100% coverage, but with multiple executions, the tendency is for test code coverage to increase.

5. RECYCLE FAILED SEEDS

5.1 CONTEXT

Used to introduce regression tests into non-deterministic algorithms and increase test code coverage. One of the biggest problems in non-deterministic algorithms is debugging, due to its random behaviour, which makes the reproduction of an error a complex task. This characteristic makes it impossible to apply certain modern techniques in software quality, for example, regression tests. The main goal of regression testing is to provide confidence that the newly introduced changes do not generate errors on existing and/or unchanged parts of the software system [Yoo and Harman 2009].

This is the most important patterns, in addition to completing the solution allows running the tests with the same entries that failed previously. This pattern makes use of the main ideas of the approaches used in Regression Test Selection (RTS) (More information in session 7) and applies it in this pattern.

5.2 PROBLEM

How to repeat and debug the execution of a failed test by promoting regression tests?

5.3 FORCES

—It is difficult to re-run a failed test again, with the same decisions that you made earlier.

—Difficulty in ensuring new code changes did not insert errors in what worked before.

5.4 SOLUTION

Save the seeds from failed tests and run new tests with these seeds to assert if changes have not injected new errors.

As a complete solution for tests of non-deterministic algorithms, as well as the ART strategy (Adaptive Random Testing) (More information in session 7), this article proposes the division of the input domain into two sets. However, unlike ART, the first set of the input domain is only a projection of the actual test cases, the second set are the test cases that have presented errors in previous implementations.

To get the first set of the input domain, use the pseudo-random generations as presented in the Re-test With Different Seeds pattern. To get the second set of the input domain, save the seeds used in failed tests. Select the seeds from the failed tests, and perform new tests with these seeds, inducing the non-deterministic algorithms to repeat the same paths and behaviors that led to failure. Like the Re-test With Different Seeds pattern, this pattern is focused on increasing code coverage, however, it is suggested to use only the seeds that failed to optimize the execution time, maintaining the regression capacity.

Figure 5 demonstrates how to use the Recycle Failed Seeds pattern. To save the seeds, verify that the characteristics have not been met, which indicate a failure, and save the seeds for later use. Still in Figure 5, the *saveFailureSeed* function represents the logic to save the seed. The seed can be saved in memory, in file or even in database, depending on the need of the software system.

Inspect Figure 5 and verify that the second part of the input domain is represented by a loop in with the failed seeds. Inside of the loop, note that the input values are generated through a pseudo-random mechanism that receives the seed that failed as a parameter, which then reproduces the same inputs, inducing the same behavior in the test. Remember that this mechanism is present in almost all programming languages.

```
@Test
private void testFixedCharacteristics(){
  int n = 2;

  //First set of the input domain.
  //Number of repetitions.
  int numberRepetitions = 100;
  for(int i=1;i<=100;i++){
    //New seed is introduced based on the computer clock.
    long new_seed = System.currentTimeMillis();
    Random rSeed = new Random(new_seed);
    int[] result = ArrayFactory.generateArrayBasedRandomSeedWithSumZero(rSeed, n);

    if(assertValueInterval(result, n)==false){
      saveFailureSeed(new_seed);
    };
    if(assertSumEqualZero(result, n){
      saveFailureSeed(new_seed);
    };
  }

  //Second set of the input domain.
  long[] allSeeds = getAllFailuresSeeds();
  for(int i=0;i<=allSeeds.length-1;i++){
    //Get failed seed.
    long fail_seed = allSeeds[i];

    //A pseudo-random input is generated based on failed seed
    Random rSeed = new Random(fail_seed);
    int[] result = ArrayFactory.generateArrayBasedRandomSeedWithSumZero(rSeed, n);

    assertValueInterval(result, n);
    assertSumEqualZero(result, n);
  }
}
```

Fig. 5. Recycle Failed Seeds Algorithm.

5.5 CONSEQUENCES

—The possibility of executing a fail test allows for a regression test;

—The possibility of executing a test with the input parameters that generated an error allows for the debugging of the code;

—Ensures that it works for previously failed cases;

—It does not guarantee 100%, but represents an evolution compared to the use of the Reply pattern, besides which with multiple executions the code coverage will always be greater.

6. RETEST FRAMEWORK EXAMPLE

A practical example of use is the Retest framework, that means Random Engagement for Test, and aims to extend JUnit framework for Java. [Ivo 2017] The Retest framework provides a mechanism for testing nondeterministic algorithms developed in Java plataform, providing a tool for its users to manage the seeds used to generate random data in the algorithm being tested. The ReTest framework is open-source and can be found at https://github.com/andreivo/retest.

Remember the motivating example: suppose you have to create an algorithm that, given an N value, should return a V vector with N integers between -10 and 10, and that the sum of those integers is zero. The trivial case, $N = 1$, is deterministic, $V = \{0\}$, so you only need to test once. But from $N > 1$, multiple vectors may be the correct answer. If $N = 2$, V can assume $\{-1, 1\}$ or $\{-2, 2\}$ or $\{-6, 6\}$, and so on, and all of them would be right. The larger the value of N, the greater the quantity of different correct vectors.

Instead of checking if the vector is equal to one of the expected responses, it is verified if: (a) the vector indeed contains N integers, (b) all numbers within the vector are between -10 and 10, and finally, (c) if the sum of the numbers is equal to zero. Regardless of the numbers that are within the vector, if it passes these three tests, the answer is correct. Moreover, because the answer is checked in three steps, if any of the tests fail, it is possible to identify which aspect of the response is not consistent with expectations. Figure 6 shows the implementation of the assertion algorithm. This allows the developer to already notice what can be wrong in the code.

The ReTest Framework suggests a function be created for characteristics assertion of the algorithm. The use of the Deterministic Characteristic Assertion Pattern is recommended, as shown in Figure 6.

```java
private void assertElements(int[] arr, int arraySize) {
  int result = 0;
  //verify if all
  for (int i = 0; i < arraySize; i++) {
    assertTrue(arr[i] >= -10 && arr[i] <= 10);
    result = result + arr[i];
  }

  //verify the sum
  assertEquals(0, result);
}
```

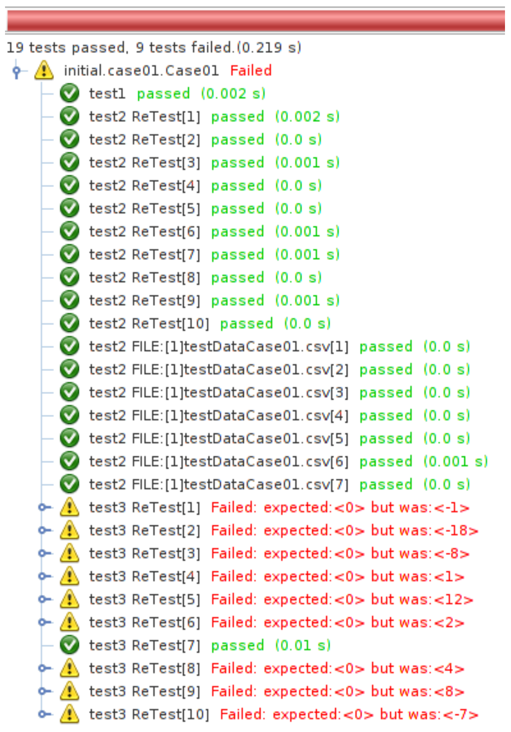
Fig. 6. ReTest Framework using the Deterministic Characteristic Assertion Pattern.

The ReTest Framework also implements the Re-test With Different Seeds and Recycle Failed Seeds patterns, as can be seen in Figure 7. The Re-test With Different Seeds pattern is represented by the @ReTest annotation, which receives, as a parameter, the number of reps of the test. The @SaveBrokenTestDataFiles and @LoadTestFromDataFiles annotations are used to implement the Recycle Failed Seeds pattern, so when a test fails, the seed is automatically saved on file to be used in the next run of tests.

```
@Test
@ReTest(10)
@SaveBrokenTestDataFiles(filePath = "/tmp/dataTest.csv")
@LoadTestFromDataFiles(filePath = "/tmp/dataTest.csv")
public void test2(@RandomParam Random r) {
  int n = 2;
  int[] result = ArrayFactory.generateArrayWithSumZero(r, n);
  assertElements(result, n);
}
```

Fig. 7. ReTest Framework using the the Re-test With Different Seeds and Recycle Failed Seeds pattern.

Figure 8 shows the interface of JUnit with the ReTest Framework. The figure shows the TDD implementation for the motivation example.



```
19 tests passed, 9 tests failed.(0.219 s)
initial.case01.Case01  Failed
    test1  passed  (0.002 s)
    test2 ReTest[1]  passed  (0.002 s)
    test2 ReTest[2]  passed  (0.0 s)
    test2 ReTest[3]  passed  (0.001 s)
    test2 ReTest[4]  passed  (0.0 s)
    test2 ReTest[5]  passed  (0.0 s)
    test2 ReTest[6]  passed  (0.001 s)
    test2 ReTest[7]  passed  (0.001 s)
    test2 ReTest[8]  passed  (0.0 s)
    test2 ReTest[9]  passed  (0.001 s)
    test2 ReTest[10]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[1]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[2]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[3]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[4]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[5]  passed  (0.0 s)
    test2 FILE:[1]testDataCase01.csv[6]  passed  (0.001 s)
    test2 FILE:[1]testDataCase01.csv[7]  passed  (0.0 s)
    test3 ReTest[1]  Failed: expected:<0> but was:<-1>
    test3 ReTest[2]  Failed: expected:<0> but was:<-18>
    test3 ReTest[3]  Failed: expected:<0> but was:<-8>
    test3 ReTest[4]  Failed: expected:<0> but was:<1>
    test3 ReTest[5]  Failed: expected:<0> but was:<12>
    test3 ReTest[6]  Failed: expected:<0> but was:<2>
    test3 ReTest[7]  passed  (0.01 s)
    test3 ReTest[8]  Failed: expected:<0> but was:<4>
    test3 ReTest[9]  Failed: expected:<0> but was:<8>
    test3 ReTest[10]  Failed: expected:<0> but was:<-7>
```

Fig. 8. ReTest Framework TDD execution example.

7. RELATED WORK

As an alternative to the Re-test With Different Seeds pattern, the ART strategy (Adaptive Random Testing) was presented as a hybrid strategy between random and guided tests [Chen et al. 2005]. It divides the input domain between two sets, cases already tested and cases to be considered. The choice of the next test to be performed is made by calculating the distance between some candidate of the second set, and the cases of the first set. The case farthest from the first set will be chosen. For our problem though, this strategy is not satisfactory. The input data for the case study are graphs, which may contain from 30 to more than 1000 vertices, and all edges that exist between them. The space for storing the set of tests already performed (graphs of various sizes), and the consuming time to calculate the distances between them and the candidate cases, are not a good alternative to combining with the fast cycles of automated tests, like the TDD technique. Therefore, it is not possible to use ART, and the Re-test With Different Seeds pattern proposes instead the use of random testing to increase the test code coverage.

Following the main idea of the technique of Regression Test Selection (RTS) the Recycle Failed Seeds pattern makes the selection of a subset of tests mainly to improve execution performance. Regression testing may become very large during the process of integration tests. Thus, it is unpractical and less efficient to re-run all the tests for each modification [Xiaowen 2013]. Because of these difficulties, an alternative approach to the Regression Test Selection (RTS) technique has been proposed in [Agrawal et al. 1993], [Chen et al. 1994], [Harrold and Souffa 1988], [Hartmann and Robson 1990] and [Rothermel and Harrold 1997]. This approach recommends that only a subset of the tests be run. This subset must be selected according to the need and characteristics of the software system being developed [Kim et al. 2000].

8. **CONCLUSION**

The idea for the technique presented in this paper arose when it was observed that the traditional automated testing technique would not be appropriate to help develop a non-deterministic algorithm. It would be impractical, as said before, to check and deterministically create all calls to the choice function that occur during execution, besides being impractical to verify all cases of the input domain, considering it is infinite. The infinite input domain also makes it impossible for our technique (or any technique in literature) to ensure that all cases are verified, but the constant running of tests with different seeds increases the coverage. Storing the seeds that caused failing tests creates a set of entries that can then be used in regression testing. Therefore, old failures are always being checked for again.

The development of a non-deterministic algorithm itself is a complex task. To do it without the assistance of a development strategy would only make the task even more difficult. The technique presented in this article allows the developer to work with more confidence, since the tests and the constant choice of different seeds explore the developed code, verifying the correctness of the answers. In addition to that, when a failure occurs, the technique allows it to be stored and checked for later, so that the error does not happen again.

REFERENCES

AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. 1993. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*. ICSM '93. IEEE Computer Society, Washington, DC, USA, 348–357.

BARESI, L. AND YOUNG, M. 2001. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A. August.

CHEN, T. Y., KUO, F.-C., MERKEL, R. G., AND TSE, T. H. 2010. Adaptive random testing: The art of test case diversity. *J. Syst. Softw. 83,* 1, 60–66.

CHEN, T. Y., LEUNG, H., AND MAK, I. K. 2005. *Adaptive Random Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 320–329.

CHEN, Y.-F., ROSENBLUM, D. S., AND VO, K.-P. 1994. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. IEEE Computer Society Press, Los Alamitos, CA, USA, 211–220.

DUNLOP, D. D. AND BASILI, V. R. 1982. A comparative analysis of functional correctness. *ACM Comput. Surv. 14,* 2, 229–244.

ELVA, R. 2013. Detecting semantic method clones in java code using method ioe-behavior. Ph.D. thesis, University of Central Florida.

FLOYD, R. W. 1967. Nondeterministic algorithms. *J. ACM 14,* 4, 636–644.

FREEMAN, S., MACKINNON, T., PRYCE, N., AND WALNES, J. 2004. Mock roles, objects. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 236–246.

HAMLET, R. 2002. *Random Testing*. John Wiley & Sons, Inc.

HARROLD, M. J. AND SOUFFA, M. L. 1988. An incremental approach to unit testing during maintenance. In *Proceedings. Conference on Software Maintenance, 1988.* 362–367.

HARTMANN, J. AND ROBSON, D. J. 1990. Techniques for selective revalidation. *IEEE Software 7,* 1, 31–36.

IVO, ANDRÉ A. S.AND GUERRA, E. M. 2017. *ReTest: Framework for Applying TDD in the Development of Non-deterministic Algorithms*. Springer International Publishing, Cham, 72–84.

KIM, J.-M., PORTER, A., AND ROTHERMEL, G. 2000. An empirical study of regression test application frequency. In *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. ACM, New York, NY, USA, 126–135.

MACKINNON, T., FREEMAN, S., AND CRAIG, P. 2000. Endo-testing: Unit testing with mock objects.

ROTHERMEL, G. AND HARROLD, M. J. 1997. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6,* 2, 173–210.

XIAOWEN, L. 2013. Research on regression testing methods for industry applications.

YOO, S. AND HARMAN, M. 2009. Regression Testing Minimisation, Selection and Prioritisation - A Survey. Tech. Rep. TR-09-09, King's College London. October.

Received May 2017; revised July 2017; accepted July 2017
October 2017

PLoP'17, OCTOBER 22-25, Vancouver, Canada. Copyright 2017 is held by the author(s). HILLSIDE 978-1-941652-06-0