A Novelty Search Approach for Automatic Test Data Generation

Mohamed Boussaa, Olivier Barais, Gerson Sunyé and Benoît Baudry
Inria/IRISA Rennes, France
Email: {mohamed.boussaa, olivier.barais, gerson.sunye, benoit.baudry}@inria.fr

***Abstract*—In search-based structural testing, metaheuristic search techniques have been frequently used to automate the test data generation. In Genetic Algorithms (GAs) for example, test data are rewarded on the basis of an objective function that represents generally the number of statements or branches covered. However, owing to the wide diversity of possible test data values, it is hard to find the set of test data that can satisfy a specific coverage criterion. In this paper, we introduce the use of Novelty Search (NS) algorithm to the test data generation problem based on statement-covered criteria. We believe that such approach to test data generation is attractive because it allows the exploration of the huge space of test data within the input domain. In this approach, we seek to explore the search space without regard to any objectives. In fact, instead of having a fitness-based selection, we select test cases based on a novelty score showing how different they are compared to all other solutions evaluated so far.**

I. Introduction

In general, manually creating test cases for testing software systems is time consuming and error-prone, making necessary the automation of this process. In fact, metaheuristic search techniques such as Genetic Algorithms (GAs) are frequently used in order to automate the test data generation process and gather relevant test cases through the wide search space [1], [2]. These techniques are especially applied for structural white-box testing. For coverage-oriented approaches, applying Evolutionary Algorithms (EAs) to test data generation has been focused on finding input data for a specific path of the program in accordance with a coverage criterion (e.g., longest path executed). The problem with coverage-oriented approaches is that search-based techniques cannot exploit the huge space of possible test data. In fact, some structures of the system cannot be reached since they are executed only by a small portion of the input domain. Applying GAs for test data generation consists in searching for relevant test data according to an objective function that tries for example to maximize the number of statements or branches covered. The use of a fitness function as a coverage criterion to guide the search to detect relevant test data usually create many local optima to which search may converge. Thus, if the relevant test data, that could coverage the longest path of the program, lie far from the search space defined by the gradient of the fitness function, then some promising search areas may not be reached. The issue of premature convergence to local optima has been a common problem in GAs. Many methods are proposed to avoid this problem [3], [4]. However, all these alternatives use a fitness-based selection to guide the search.

In this paper, we introduce the use of Novelty Search (NS) algorithm to the test data generation problem. In this approach, we seek to explore the search space of possible test input values without regarding to any objectives (there is no fitness function). In fact, instead of having a fitness-based selection, we rather select test cases based on a novelty score showing how different they are compared to all other test data evaluated so far. So during the evolutionary process, we use to select test data that remain in sparse regions of the search space in order to guide the search through novelty. We choose the statement coverage metric as a coverage criterion to our NS-based test data generation.

The primary contribution of this paper can be summarized as follows: the paper introduces a novel formulation of the test data generation problem using NS and, to the best of our knowledge, this is the first paper in the literature to use NS algorithm to generate test data.

The paper is organized as follows: section II describes the related work. The approach overview and the NS adaptation are presented in Section III. Finally, concluding remarks and future work are provided in Section IV.

II. Related work

A. Generating Test Data

Most forms of automatic test data generation have been focused on finding specific input values that meet a specific testing criteria. In search-based test data generation, Harman and McMinn [5] conducted a large empirical study that compares the behavior of both global and local search-based optimization on real-world problems. The results show that the use of EAs is suitable in many cases. However, it can be outperformed by simpler search techniques. They presented as well, a Memetic Algorithm (Hybrid Algorithm) that combines the global and local search.

Furthermore, a lot of research work have been conducted in the field of structural code coverage. In the work of Roper for example [1], GAs are used to generate test data based on the number of structures executed in accordance with a coverage criterion. Whereas, Watkins [2] attempts to obtain full path coverage for programs.

For test case selection strategies, Chen et al. [6] presented a synthesis of the most important results in Adaptive Random Testing (ART) (an extension of Random Testing). They outlined the importance of diversity in test case selection. In fact, they argue that diversity among test cases should be rewarded

because failing test cases tend to be clustered in contiguous regions of the input domain. The success of ART motivated us to introduce a new technique that maximizes input diversity.

B. Novelty Search

In the literature, EAs are often applied to the test data generation problem [7]. These techniques use basically a fitness function to guide the search e.g., gather fittest solutions over generations. These techniques are good since they reward individuals with high score but they do not favor diversity and the search may converge to many local optima [3], [4]. The idea of NS, introduced by Lehman and Stanley in 2008 [8], represents an alternative solution for this issue. In fact, individuals in an evolving population are selected based on how different they are compared to other solutions evaluated so far. They also argue that objective fitness functions can be deceptive, leading the evolutionary search to local maxima rather than towards the goal. In contrast, NS ignores the objective and simply searches for novel behavior, and therefore cannot be deceived. So mainly, NS acts like GAs. However, NS needs extra changes. First, a measure of individuals behavior. This depends on the context of the search and the way we represent individuals. Then, a new novelty metric, which rewards individuals with different behavior from past-discovered solutions. Finally, an archive must be added to the algorithm which is a kind of a data base that remembers individuals that were highly novel when they were discovered in past generations. NS has been often evaluated in deceptive tasks and applied to evolutionary robotics (in the context of neuroevolution) [9], [10].

III. Novelty Search for Test Data Generation

A. Approach Overview

Our test data generation framework aims to fully automate the test data generation process and avoid any tester's implication. We aim to use a new optimization evolutionary technique, namely Novelty Search, to the test data generation problem.

To automate the test data generation process, we have considered that our System Under Test (SUT) is like a gray/semi-transparent box in where the internal structure is partially known. Thus, we can design test cases through the exposed interfaces and conduct a code coverage analysis from the general structure of our target SUT. For example, within *apache.commons.math* library, Methods Under Test (MUTs) are accessed through some specific Java interfaces. Each interface belongs to a sub-package of the whole library and exhibits a set of methods. Our test data generation framework will rely on this concept to generate automatically test cases. In fact, as shown in Figure 1, starting from an input interface and a source code package, the testing framework is able to: (1) automatically generate sequences of method invocation, (2) generate relative test data, (3) execute test cases on target Classes Under Test (CUTs), and then (4) analyze the code coverage. The process is iterated until a termination criterion is met (e.g., number of iterations).

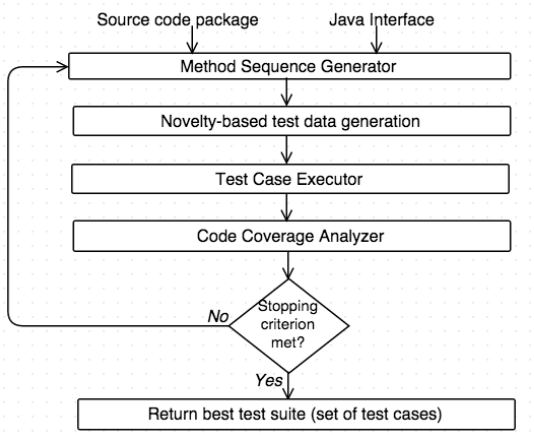

Fig. 1. Approach overview

The main task of our proposal is to use an alternative approach for test data generation based on the NS algorithm. To do so, we adapt the general idea of NS, presented earlier in the related work, to the test data generation problem. In fact, instead of using a fitness function to evaluate generated test cases, we define a new measure of novelty to maximize. We replace, as well, the fitness-based selection (for fittest test cases) with a novelty-based one. This may favor the diversity of generated test data. In addition, although we are exploring the search space without regarding to any objective, we keep a measure of statements coverage for each set of test cases so that, by the end of the evolutionary process, we can gather only test cases with high coverage value. Finally, we add an archive that acts as a memory of previously generated test cases. This archive is used to calculate the novelty metric. Otherwise, we keep the same logic as GAs, namely the crossover and mutations operators. In the following, we detail the main tasks of our automatic test data generation framework:

1) Method Sequence Generator: The method sequence generator is used to automatically generate the test scenario. It takes as an input a Java interface and a source code package. The generated sequence represents the methods names and their parameters types as declared in the Java interface. Indeed, we keep the same number and order of methods signatures as defined in the interface. In this way, we test each method of the target classes in isolation without considering dependencies among methods.

2) Novelty-based Test Data Generation: To generate test cases, we produce input data that will be passed as arguments to the MUTs. Depending on parameters types, we generate randomly input values for Java primitive types and strings. In fact, we did not define any constraint on test data values and we have considered all possible input values for each data type. This process applies the NS approach to generate test data.

3) Test Case Executor: Initially, the test case executor explores the source code package and catches all classes (CUTs) implementing the input interface. Once this is done, we execute automatically our set of test cases on top of these classes except for abstract classes.

4) Code Coverage Analyzer: Our testing framework defines an analyzer able to instrument and analyze all Java classes within a source code package. Thereby, after the execution of a set of test cases, instrumented code is analyzed and a coverage value is assigned to each set of test cases. For instance, we check the number of executed statements. We use *Jacoco*[1] as a tool to instrument classes and compute the coverage value.

B. Novelty Search Adaptation

In this section, the main contribution is presented, namely, a method for generating automatically test data using NS. In order to ease the understanding of this new approach, we first describe the pseudo code of our adaptation. Then, we present the solution structure and the novelty function used to evaluate solutions.

Algorithm 1: Novelty search for test data generation

Require: Java interface I
Require: Source code package Pack
Require: Number of iterations N
Require: Population size PopSize
Require: Coverage threshold minCoverage
Require: Novelty threshold T
Require: Limit L
Require: Nearest neighbors k
Ensure: Set of relevant test cases bestSolutions

```
1:  targetClasses ← loadAllMethods(I, Pack)
2:  testCases ← generateTestCases(I)
3:  repeat
4:      testSuite ← generateTestData(testCases)
5:      P ← setOf(testSuite)
6:      for testSuite ∈ P do
7:          for testCase ∈ testSuite do
8:              coverage ← execute(testCase, targetClasses)
9:          end for
10:         noveltyMetric ←
            distFromkNearest(testSuite, archive, P, k)
11:         if noveltyMetric > T then
12:             archive ← archive + testSuite
13:             selectedTS ← selectedTS + testSuite
14:         end if
15:         if coverage ≥ minCoverage then
16:             bestSolutions ← bestSolutions + testSuite
17:         end if
18:     end for
19:     P ← generateNewPopulation(selectedTS)
20:     generation ← generation + 1
21: until generation = N
22: return bestSolutions
```

Algorithm 1 describes the overall idea of our NS adaptation for test data generation: The algorithm takes as input a Java interface and a source code package. We initiate the number of iterations N, the population size and the minimum coverage value. This latter defines the threshold of covered statements that should be reached. Test cases that exceed this threshold are automatically added to the set of relevant test cases. Since we want to compare our NS approach to the fitness-based and random approaches, we are going to set the minimum coverage value based on the maximum coverage value reached by these two approaches. So that, we can record only test cases that may overcome those generated by fitness-based and random approaches. Same thing for the novelty threshold T that defines the threshold for how novel a test suite has to be before it is added to the archive. In addition, we define a maximum size limit for the archive L and a k number that will be used in calculating the distance from k-nearest neighbors. k is a fixed parameter that is determined experimentally.

First, we load the set of classes (CUTs) that implement the given interface I (Line 1). Before starting the test data generation process, we define our set of test cases that will be used to generate test data (Line 2). Line 3-21 encode the main NS loop, which searches for the best set of test cases. During each iteration, we generate a new population that stands for a set of test suites. Then, we generate random test data (Line 4) and we execute our set of test cases on target classes (Line 7-9). The coverage value is computed and assigned to each solution (the executed test cases). This value corresponds to the total number of executed statements by the total number of statement being in the source package.

In the same way, for each set of test cases in the population, we calculate the average distance from its k-nearest neighbors (Line 10). If the novelty is sufficiently high (higher than the given threshold T), then the set of test cases is selected (Line 13) and entered into the permanent archive (Line 12). Genetic operators (mutation and crossover) are applied later to these selected test cases in order to produce offspring individuals and fulfill the next population (Line 19). Finally, we save relevant test cases that reach a coverage value higher than the minimum coverage threshold defined initially (Line 15-17).

1) Solution Representation: For our case study, a candidate solution represents the set of methods declared in the input interface (see Figure 2). Thereby, we represent this solution as a vector where each dimension has a method name, a list of parameters types and a list of test data. The test scenario remains the same for all individuals. In fact, we always run the same number of test cases with the same order as defined in the Java interface. This is important since the order in which test cases are executed may affect the rate of code coverage [11]. Therefore, we may focus only on the test data generation problem instead of focusing on the test case prioritization issue. Additional information should be added to each candidate solution such as the CUTs where the test cases should be executed and also required constructors needed to create objects from the CUT. In fact, we save for each set of test cases the list of CUTs where the test cases have to be executed and the list of constructors required for each CUT. The constructor representation is the same as test cases. The test case executor presented earlier in the sub-section A.3 chooses arbitrary a constructor from the list of constructors

[1] http://www.eclemma.org/jacoco/index.html

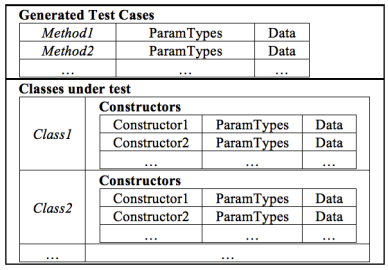

Fig. 2. SOLUTION REPRESENTATION

relative to the target CUT in order to instantiate objects and run test cases.

2) Novelty Metric: The Novelty metric measures the distance of a test suite (solution) to all other test suites in the current population and to those that were discovered in the past. The distance between two test suites is computed as a Manhattan distance between the input parameter values of all methods tested in the test suite. Formally, we define this distance as follows:

$$distance(t1, t2) = \sum_{i=1}^{m} \sum_{j=1}^{p} |t1(M_i, P_j) - t1(M_i, P_j)| \quad (1)$$

where $t1$ et $t2$ are two selected test suites (solutions), m is the number of methods composing the test suite, p is the number of parameters composing a method. The couple (M_i, P_j) returns the j^{th} parameter value of the i^{th} method M relative to a test suite ($t1$ or $t2$). Since we are using Java primitive types for test data such as floats, integers, doubles, etc, it is easy to calculate this distance for numerical parameters values. However, for string data types we use the Levenshtein Algorithm[2] to measure the strings distance. Furthermore, we have normalized this distance in the range [0-100].

To measure the sparseness of a test suite, we will use the previously defined distance to compute the average distance of a test suite to its k-nearest neighbors. In this context, we define the novelty metric of a particular solution as follows:

$$NM(S) = \frac{1}{k} \sum_{i=1}^{k} distance(S, \mu_i) \quad (2)$$

where μ_i is the i^{th} nearest neighbor of the solution S within the population and the archive of novel individuals.

In order to test the applicability of our approach for test data generation, we conduct initial implementation and experimentation of our NS Algorithm based on Java Libraries. Thus, we have opted to generate test cases based on *apache.commons.math* interfaces. For instance, we choose the '*UnivariateFunction*' interface from the sub-package *apache.commons.math.analysis* to generate test data. This interface is implemented by 42 classes within *apache.commons.math*. We are going to target only classes within the sub-package *apache.commons.math.analysis.function*. To compute the novelty, we used *k*=20 (*PopSize/5*) and *T* = 50 (*max(NM)/2*). The population size is fixed to 100 and the number of generations to 1000. In this way, the algorithm performs 100000 evaluations. Since we haven't yet the minimum coverage value (the max fitness value from GAs), we are trying with this initial implementation, to record the maximum coverage value that may be reached by applying the NS. Initial project and results can be found at the project website[3].

IV. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new search-based approach for test data generation. In our NS adaptation, we try to exploit the large search space of input values and catch relevant test cases that may cover as much as possible the executed statements. However, the limitation of this approach lies in the use of continuous data to compute the novelty score. If we assume that the input values may have only certain values or categorical data, the distance score will not be straightforward and we may use other similarity measures for categorical data.

As future work, we aim to conduct an empirical evaluation of our NS approach by comparing it to fitness-based and random approaches. In addition, we can optimize our approach by adding the diversity as an addition goal to a traditional objective driven approach to form a multi-objective optimization problem. Finally, since we are testing gray-box systems, we can also apply this approach for black-box testing. In this case, we will be able to measure some non-functional properties such as memory usage and CPU consumption.

REFERENCES

[1] M. Roper, "Computer aided software testing using genetic algorithms," *10th International Quality Week*, 1997.

[2] A. L. Watkins, "The automatic generation of test data using genetic algorithms," in *Proceedings of the 4th Software Quality Conference*, vol. 2, 1995, pp. 300–309.

[3] W. Banzhaf, F. D. Francone, and P. Nordin, "The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets," in *Parallel Problem Solving from NaturePPSN IV*. Springer, 1996, pp. 300–309.

[4] C. Gathercole and P. Ross, "An adverse interaction between crossover and restricted tree depth in genetic programming," in *Proceedings of the 1st annual conference on genetic programming*. MIT Press, 1996, pp. 291–296.

[5] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 226–247, 2010.

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[7] M. Harman, L. Hu, R. M. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal." in *GECCO*, 2002, pp. 1359–1366.

[8] J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in *ALIFE*, 2008, pp. 329–336.

[9] S. Risi, C. E. Hughes, and K. O. Stanley, "Evolving plastic neural networks with novelty search," *Adaptive Behavior*, vol. 18, no. 6, pp. 470–491, 2010.

[10] P. Krčah, "Solving deceptive tasks in robot body-brain co-evolution by searching for behavioral novelty," in *Advances in Robotics and Virtual Reality*. Springer, 2012, pp. 167–186.

[11] P. R. Srivastava, "Test case prioritization," *Journal of Theoretical and Applied Information Technology*, vol. 4, no. 3, pp. 178–181, 2008.

[2]http://www.levenshtein.net/

[3]http://goo.gl/T3U0v2