Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components

Benoit Baudry*, Franck Fleurey**, Jean-Marc Jézéquel* and Yves Le Traon*
* IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Benoit.Baudry, jezequel, Yves.Le_Traon}@irisa.fr
** IFSIC, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
** franck.fleurey@ifsic.univ-rennes1.fr

Abstract

In this paper, we present several complementary computational intelligence techniques that we explored in the field of .Net component testing. Mutation testing serves as the common backbone for applying classical and new artificial intelligence (AI) algorithms. With mutation tools, we know how to estimate the revealing power of test cases. With AI, we aim at improving automatically test cases efficiency. So, we looked first at genetic algorithms (GA) to solve the problem of test. The aim of the selection process is to generate test cases able to kill as many mutants as possible. Then, we propose a new AI algorithm that fits better to the test optimization problem we called bacteriological algorithm (BA): BAs behave better that GAs for this problem. However, between GAs and BAs, a family of intermediate algorithms exists: we explore the whole spectrum of these intermediate algorithms to determine whether an algorithm exists that would be more efficient than BAs.: the approaches are compared on a .Net system.

1. Introduction

In the .Net framework, few attention has been paid for improving the internal consistency and robustness of components. This consistency can be greatly improved when the code can benefit from design specification: this is the case with contracts, that can be seen as structural/functional properties derivable into executable assertions (invariant properties, pre/postconditions of methods). We present a testing for trust methodology particularly adapted to a design-by-contract approach [1] that uses mutation as a backbone technique. Mutation analysis consists in systematically introducing faults in the component under test to produce *mutants* (modified version of a component). It is then possible to qualify a set of test cases by computing a *mutation score* corresponding to the proportion of mutants the set can detect. This approach is especially well adapted for .Net components, where we can benefit from the interoperable platform. The challenge for applying this technique resides in the difficulty of injecting faults at the Intermediate Language (IL) level, that may correspond to a code generated from a faulty source code. We detail in the following how this issue can be overc ame.

While it is reasonable to expect from the tester the systematic delivery of test cases, it is an arduous task to expect these test cases to be 100% efficient, in terms of mutation score. The work presented in this paper focuses on automating the test cases improvement step. We present here the adaptation of genetic algorithms to this context, and analyze the results obtained with a case study: optimizing test cases for a C# parser in the .Net framework. Because the test case optimization is more an adaptation process than an evolution process, we propose bacteriological algorithms, taken from the bacteria adaptation phenomenon. Results show that this new algorithm provides a better optimization of test cases for the case study. Since BA differs from GA by the memorization of best test cas es and by the unbounded size of the individuals, an infinite set of intermediate algorithms exists between BA and GA. So, we explored all the spectrum of possible algorithms from genetic to bacteriological algorithms.

The rest of this paper is organized as follows. Section 2 opens with a brief summary about mutation analysis, its application to .Net and the design for trust methodology for .Net. Section 3 compares genetic and bacteriological algorithms. At last, section 4 defines the family of algorithms between genetic and bacteriological ones: the study aims at determining whether an intermediate algorithm can be more efficient than bacteriological algorithms.

2. Trusting components and Mutation

We first recall the mutation analysis process and how it has been adapted to generate and validate test cases for a component. Then we summarize a methodology for developing trustable software components.

2.1. Mutation Testing

Mutation testing is a technique which was first designed to create effective test data, with an important fault revealing power [2]. It has been originally proposed in [3], and consists of creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test cases that distinguish the program from all its mutants. In practice, faults are modeled by a set of *mutation operators* where each operator represents a class of software faults. A mutant is created by the insertion of one error in the original program. When generating mutants from a set of mutation operators, one might create *equivalent mutants* (no input data can distinguish the mutant from the original component). A *mutation score(MS)* is associated with the test cases set to measure its effectiveness in terms of *percentage* of the revealed non-equivalent mutants.

2.2. Mutation analysis for .Net

It seems that mutation analysis has never been applied to .Net components. At first glance, one may consider that a mutation tool should inject faults at the IL level, in order to have a common backbone for any high-level language. This consideration is contradictory with the competent programming hypothesis, since faults at IL level do not necessarily correspond to real faults. To overcome this problem, we need to bridge the gap between high-level faults and the way they are compiled in IL. For the objective of this paper, we reverse the reasoning and we produced the faulty IL code from faulty C# code. That means that the mutation tool is dedicated to C# source code. Moreover, since decompilers from IL to C# are becoming available working at C# level (http://www.remotesoft.com/salamander/) will guarantee that faults have a realistic semantic.

When applying mutation analysis at component/system level scale problems appear: prohibitive number of mutants, long execution time, difficulty in determining equivalent mutants.

2.3. Design for trust methodology

Concerning component-based systems, we propose a testing-for-trust approach, that checks the consistency of a component's three facets, i.e., specification, implementation and tests. This method uses a *mutation analysis* to produce efficient test cases. The test cases can then detect weaknesses in the implementation and the specification. The derivation of contracts may be automated from UML (thanks to the OCL - Object Constraint Language) to any given language such as Eiffel#. Language-independent contracts should be one of the expected features offered by the .Net paradigm (http://www.codeproject.com/csharp/designbycontract.asp for a first practical contribution in this direction). As several testing frameworks now offer this feature (Craig's work http://nunit.sourceforge.net/) to components under test , test suites are defined as being an "organic" part of OO software component: the component is made *self-testable* [4]. A component is composed of its specification (documentation, methods signature, invariant properties, pre/ postconditions), one implementation and the test cases needed for testing it (its *selftests*).

From a methodological point of view, we argue that the trust we have in a component depends on the consistency between the specification (refined in executable contracts), the implementation and the test cases. The question is thus to be able to estimate this consistency. The chosen quality criteria proposed here is the mutation score. The global component design-for-trust process consists of 5 steps:

1. the programmer writes an initial selftest that reaches a given initial Mutation Score (MS).
2. the initial selftest is enhanced. We study various optimization algorithms for that purpose. The differences between output traces serves as oracle.
3. the user has to check if the tests do not detect errors in the initial program and debug them.
4. measure the contracts quality thanks to mutation testing. We use the *embedded contracts* as oracle.
5. improvement of contracts to reach an expected quality

In the remaining of the paper, we concentrate on automating step 2, by applying optimization algorithms based on classical models (genetic algorithms) and original ones (bacteriological and mixed-approach).

3. Genetic vs. bacteriological algorithms

In this section, we investigate the use of genetic and bacteriological algorithms as a pragmatic way to automatically improve the basic test cases set in order to reach a better test quality level with limited effort. The complete study is presented in [5]. The experiment is based on a component written in C# in the .NET framework. It implements a simplified parser for the C# language (32 classes). The case study has been chosen to represent the category of software that transforms input data in a given format into a new format. For instance, the same modeling of GAs can be directly used for testing software using the XML as an exchange format. To experiment genetic algorithms on this component, we generated 500 mutants. The initial population consisted of 12 test cases (individuals): the mutation score was 56%.

3.1. Genetic algorithms for test optimization

In our modeling of genetic algorithms [6], a *gene* corresponds to a *test case*. For this studied C# parser, the input data is a source file that is parsed to build a syntactic tree. Then individuals can be build, corresponding to a finite and bounded string of genes, and a set of individuals is called a population. A second criterion needs to be defined : a *fitness function* F which qualifies the individual regarding the problem we want to solve. In our case, the *fitness function* is equal to the *mutation score of the individual*. Moreover, a genetic algorithm uses three operators: reproduction, crossover, mutation. Since genes corresponds to a C# code represented by its syntactic tree, the mutation operator consists here in replacing a syntactic node in a source file by another licit node. The mutation operator thus chooses a gene at random in an individual and replaces a node in that gene by another one. Then, the genetic algorithm is computed by generating new test cases at each generation.

The experiments with genetic algorithms were not satisfactory. We had to excessively increase the mutation rate compared to usual application of genetic algorithms (mutation rates lower than 2%). With 2% of mutation rate, the best mutation score was 80%, after a long computation. Moreover, the results are not stable, the convergence is slow and one population can be more efficient from the following, due to a non-explicit memorization.

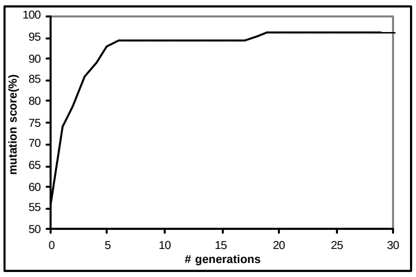


Figure 1 – The bacteriological approach

3.2. Bacteriological algorithms

The bacteriological approach aims at mutating the initial population to adapt it to a particular environment. The individuals in the population are called *bacteria*. Unlike the genetic model the bacteria can not be divided. Bacteria can only be reproduced and altered to improve the population. The new approach is fairly far from the genetic model. If we keep the analogy with biological processes, this new model is close to the "bacteriologic adaptation" [7]. Bacteria are selected for reproduction according to their mutation score. The best bacteria are *saved* and reproduced to generate a new population. The notion of individual disappears as well as reproduction and crossover operators.

Figure 1 shows results of the bacteriological approach: the test cases reach 95% of mutation score. Moreover, the algorithm is more stable than GA, and repeated experiments produce the same kind of curves. So BAs are more adapted to this test optimization problem than GAs.

4. A family of algorithms

The main parameter that differentiates a genetic algorithm from a bacteriological one is related to the notion of memorization. A bacterium (equivalent to a gene in the GA) that kills new mutants is systematically memorized. On the opposite, in the GA approach, individuals/genes are never memorized. Between these two opposite solutions, we fix a threshold for memorization and obtain the following family of algorithms. Let B be a bacterium:

```
if fitness_value(B)> threshold_value then memorize
   B
```

The category of the algorithm depends on the threshold value:

```
if threshold_value = 100 then "pure " genetic
if threshold_value = 0 then "pure" bacteriological
if 0 < threshold_value < 100 then mixed-approach
```

The question is to determine whether one of these intermediate algorithm can behave better than a BA. Two criteria are taken into account:

- Minimization of the resulting size of the set of test cases to minimize execution time. For a given bacterium size, the number of memorized bacteria is a good indicator of the final test cases set size. Figure 2 displays the impact of the memorization threshold and number of memorized bacteria.
- Minimization of the number of generations needed to obtain a top mutation score. Figure 3 presents the variation of the algorithms convergence in function of the memorization threshold.

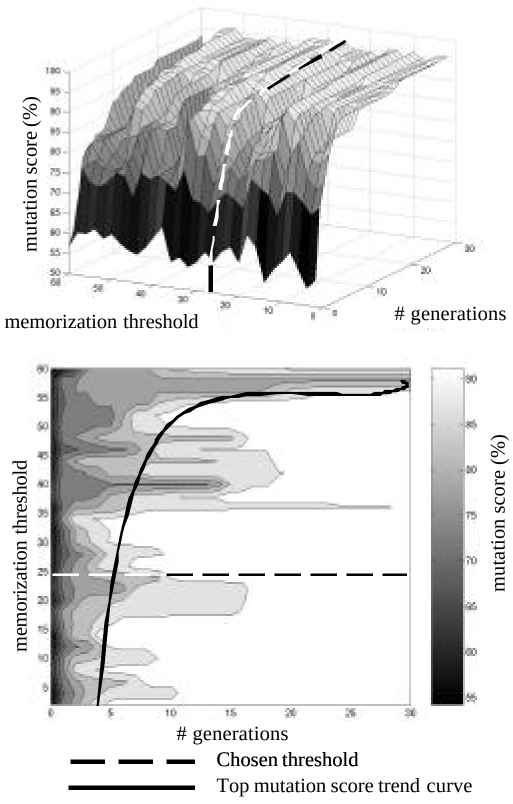
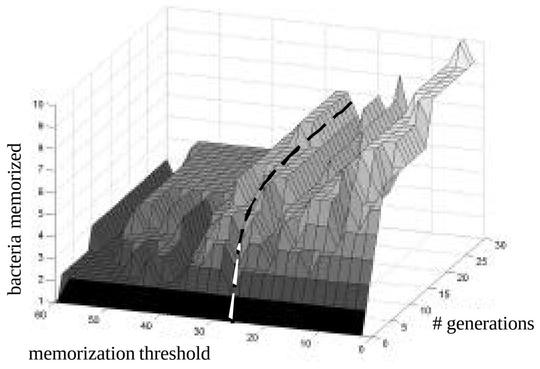


Figure 2 - Number of memorized bacteria

We explore the spectrum of possible algorithms between the GAs solution and the bacteriological one, what we call a "mixed approach". The goal is to determine if a local optimum exists, better than GAs and bacteriological opposite solutions.



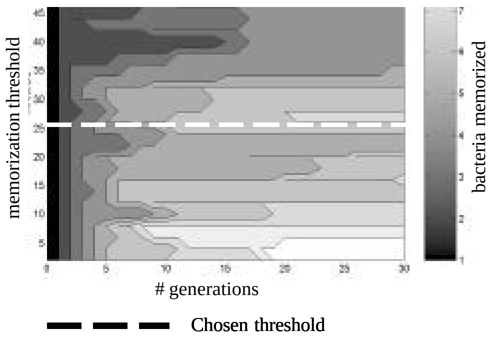


Figure 3 - Convergence speed

While Figure 2 shows that the number of memorized bacteria (and thus the set size of test cases) decreases regularly when the memorization threshold increases, Figure 3 reveals that the convergence speed decreases when the threshold is upper than 30. A possible trade-off can be determined in a range of threshold values between (approximately) 20 and 30: the final test cases set size is around 7 and the convergence speed is reasonable (around 10 generations to reach 90% mutation score). Figure 4 shows the slice for a threshold of 25. Compared to the bacteriological algorithm, the mixed-approach:

- allows the reduction of the test cases set from 10 to 7,
- the 95% mutation score is only reached at the 25th generation (20th with bacteriological – see Figure 1).

The tuning of the mixed-approach is difficult and the benefits not obvious. To our mind, the 'pure' bacteriological approach main advantage is its generality and the easiness for reuse (no parameter other than the bacterium size to be tuned) associated to high speed convergence.

5. Conclusion

The general assumption for this work is to measure the quality of test cases (the revealing power of the test cases [2]), to build trust in a component passing those test cases. In this paper, we have presented several complementary computational intelligence techniques that we explored in the field of .Net component testing: mutation testing and the whole spectrum of algorithms between GAs and BAs (bacteriological algorithms). The results are promising for the category of systems that makes systematic transformation of data from one format to another (such as XML based software).

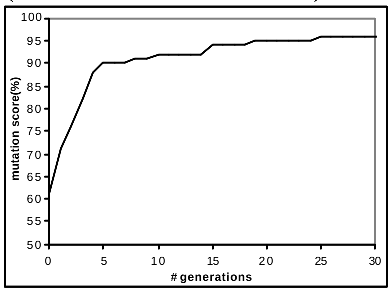


Figure 4 - The mixed-approach

Acknowledgment Many thanks to Françoise Burel, director of the "Ecosystem Functioning and Conservation Biology" lab of Rennes I University, for her helpful remarks and suggestions in the definition of the bacteriological algorithm

References

[1] B. Meyer, "Applying Design by Contract," *IEEE Computer*, vol. 25, pp. 40-51, 1992.

[2] J. M. Voas and K. Miller, "The Revealing Power of a Test Case," *Software Testing, Verification and Reliability*, vol. 2, pp. 25-42, 1992.

[3] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection : Help For The Practicing Programmer," *IEEE Computer*, vol. 11, pp. 34-41, 1978.

[4] J.-M. Jézéquel, D. Deveaux, and Y. Le Traon, "Reliable Objects: a Lightweight Approach Applied to Java," *IEEE Software*, vol. 18, pp. 76-83, 2001.

[5] B. Baudry, F. Fleurey, Y. L. Traon, and J. Jean-Marc, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," presented at ISSRE 02 (Int. Symposium on Software Reliability Engineering), Annapolis, MD, USA, 2002.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*: Addison-Wesley, 1989.

[7] M. L. Rosenzweig, *Species Diversity In Space and Time*: Cambridge University Press, 1995.

Remark: The mutation tool, the genetic and bacteriological libraries and the case study are available at http://franck.fleurey.free.fr