Automatic Test Case Optimization: A Bacteriologic Algorithm

Benoît Baudry, Franck Fleurey, Jean-Marc Jézéquel, Yves Le Traon

▶ **To cite this version:**

Benoît Baudry, Franck Fleurey, Jean-Marc Jézéquel, Yves Le Traon. Automatic Test Case Optimization: A Bacteriologic Algorithm. IEEE Software, 2005, 22 (2), pp.76-82. 10.1109/MS.2005.30 . hal-03524253

HAL Id: hal-03524253

https://hal.science/hal-03524253

Submitted on 17 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automatic Test Cases Optimization: a Bacteriologic Algorithm

Version #3

Submitted 8 November 2004

Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, Yves Le Traon

IRISA – Université de Rennes 1,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, France
{Benoit.Baudry, Franck.Fleurey, Jean-Marc.Jezequel, Yves.Le_Traon}@irisa.fr

Abstract. The quality of test cases is an important factor to estimate the confidence one can have in a component under test. This quality can be evaluated with *mutation analysis*: the quality of the test cases is evaluated by the proportion of seeded faults detected by the test cases. While the generation of a basic test cases set can be easy, improving its quality may require prohibitive effort. This paper focuses on the issue of automating the test optimization. A novel algorithm is proposed, adapted from genetic algorithms that is called a *bacteriologic algorithm* and is inspired by the biological phenomenon of *evolutionary ecology*. The approach is illustrated with test generation for a C# parser.

Keywords:

I.2.m.c Evolutionary computing and genetic algorithms

D.1.5 Object-Oriented Programming,

D.2.5.l Test design

1. Introduction

Whereas testing is extensively used to asses the quality of a software product, it is important to be able to asses the quality of the test phase itself. Indeed, the more efficient the test cases are the more testing we can perform in a given time and therefore the more confidence we can have in the software. A technique called *mutation analysis,* originally proposed by De Millo in [1], offers an interesting approach to build confidence in the test cases. This analysis consists in introducing faults in the software under test and the intuition is that test cases are good if they are able to detect these faults. Mutation analysis has been successfully applied to qualify unit test cases for OO classes [2-4], and gives the programmer interesting feed-back on the "revealing power" of his/her test cases. It also offers an estimate of how many new test cases are needed to better test a given software component.

While the generation of a set of basic test cases may be easy, improving its quality usually require prohibitive effort. Indeed, the test cases that are generally provided by the tester easily cover 50-70 % of the introduced faults, but improving this score up to 90-100 % is a time-consuming and therefore expensive task. This paper focuses on automating the test improvement stage, i.e. test optimization.

The issue of improving test cases automatically is a non-linear optimization problem. This paper introduces a novel algorithm to solve this problem. It is adapted from genetic algorithms [5] and is inspired by the biological phenomenon of *evolutionary ecology*. It is called a *bacteriologic algorithm* and is designed to generate or optimize a set of test cases. The approach is illustrated in this paper with an example based on a parser for C#. First we introduce mutation analysis, then we go on with the presentation of the bacteriologic algorithm and its application for automatic test cases improvement.

2. Case study: a parser for C#

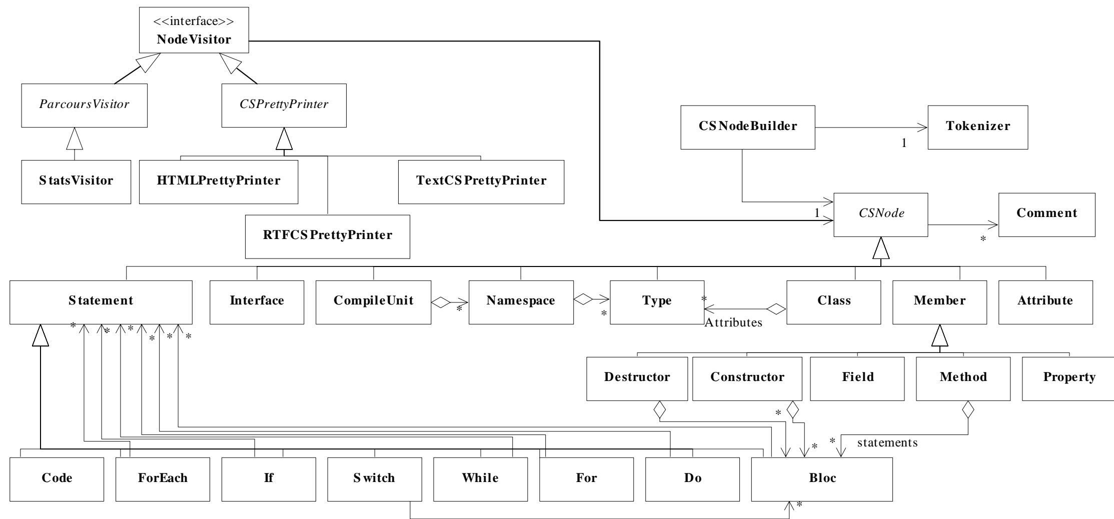

Figure 1 - Parser for the C# language

```
[1]     using System;
[2]     namespace Id_1 {
[3]       using System;
[4]       protected class Id_2 {
[5]         [AnAttribute1; AnAttribute2]
[5]         public string aField;

[6]         public ~Id_2() { } //~Id_2

[7]         [AnAttribute1; AnAttribute2]
[8]         public Id_2() { } //Id_2

[9]         [AnAttribute]
[10]        public virtual returnType aMethod (Type1 param1, Type2 param2) ;

[11]        [AnAttribute]
[12]        static Type aProperty {
[13]        get { }
[14]        set {
[15]        aVariable = aValue + 3;
[16]          for (int i=0 ; !Id_6||Id_8!=Id_3 ; i++)
[17]            {foreach (nodes n in the_tree)
[18]              {anObject.aMethod (param3, param4 );}}
                }
              }

[19]        public returnType1 aMethod2 (Type3 param5) { } //aMethod2
[20]      } //Id_2}
```

Figure 2 - Test case example for the C# parser

The algorithm presented in this paper is illustrated with a .Net component (implemented in C#) that parses C# source files [6]. This parser takes a set of C# source files as an input and builds the corresponding syntactic tree. The UML class diagram for this parser is given Figure 1. There are 32 classes in this system that can be divided in three main parts. First, the `CSNodeBuilder` class is the main class for building the syntactic tree. Second, the inheritance hierarchy under the `CSNode`, corresponds to the node types in the C# abstract syntactic tree. The third part of the diagram is the `NodeVisitor` interface and its different implementations.

These classes correspond to the application of the Visitor design pattern [7], which enables implementing different processing on the syntactic tree.

A test case for this parser is a syntactically correct C# source file like the one shown in Figure 2.

3. Mutation analysis

Mutation analysis was first designed to create effective test data with important fault revealing power [8]. It introduces faults in the component under test (CUT) to create a set of *mutants* (each of which contains a single fault). The goal is then to design a set of test cases that distinguishes the component from all of its mutants. A mutant that has been detected by a test case is said to be *killed* by the test case, otherwise it is *alive*. When generating mutants one might create *equivalent mutants*. A mutant is said to be equivalent if no input data can distinguish the output of the mutant from the output of the original component.

The quality of a test cases set can be evaluated by its *mutation score (MS).*

Mutation Score. *Let d be the number of dead mutants after applying the test cases, m the total number of mutants and equiv, the number of equivalent mutants.*

The mutation score MS for a test cases set T is defined as follows:

$$MS(T) = 100 \times \left(\frac{d}{m - equiv}\right)$$

In practice, faults are modeled by a set of *mutation operators* where each operator represents a class of software faults. For example, in [2, 9] several categories of mutation operators are proposed. In the study presented here, mutation analysis is applied on a component built with several classes. Since the number of mutants increases with the size of the component, and the execution time increases with the number of mutants (all the test cases must be executed against all the mutants), a limited number of mutation operators has to be chosen.

- LOR: each occurrence of one of the logical operators (and, or, nand, nor, xor) is replaced by a different one of the other operators; in addition, the expression can be replaced by TRUE or FALSE.
- NOR: suppresses a statement or a block of statement.

For example, on the method `accept` given below:

```
    public override void accept(NodeVisitor v) {
[1]   if (requestedMutant > -1) {
[2]      BlockNode n = (BlockNode)getMutant(requestedMutant);
[3]      if (n != null) v.visitBlockNode(n);
```

```
[4]      else v.visitBlockNode(this);}
  else v.visitBlockNode(this);
  }}
```

- Mutant for the LOR operator statement 3 is replaced by:

```
if (true) v.visitBlockNode(n);
```

- Mutant for the NOR operator: statement 2 is deleted.

Making the assumption that all classes in the component have been tested at unit level, we believe using two operators is sufficient (LOR and NOR operator defined below). This assumption is realistic after unit testing: in that case testing focuses on the interactions between units. These operators guarantee code and predicate coverage which is sufficient to cover the interactions between units.

When a set of mutant components is automatically generated with the selected mutation operators, the test cases are executed against each mutant. If the output is different from the output produced by the test case execution on the initial program, the mutant is killed by the test case. This specific oracle function is meaningful to evaluate the quality of the test case. Indeed, mutation analysis aims at checking the ability of the test cases to detect the errors that have been intentionally injected in the initial program. Thus, it aims at checking if the test cases are able to detect the difference between the initial program and the mutant. Once good test cases have been generated using mutation analysis, they are executed against the initial program to detect real errors in this program (which is the one we actually want to test).

If a mutant program is not killed by any test case, a diagnosis step determines the reason of non-detection. The mutant may be alive because a test cases are too weak or it is an equivalent mutant. In the following we introduce a novel algorithm that automates the test case enhancement phase after the diagnosis step.

4. A bacteriologic algorithm for automatic mutation score improvement

Our novel algorithm is called a bacteriologic because it is inspired by *evolutionary ecology* [10] and more particularly *bacteriologic adaptation*. Evolutionary ecology is defined as the study of living organisms within the context of their environment, with the aim of discovering how they adapt [10]. The fundamental concept of this approach is that in a heterogeneous environment it is not possible to find a single individual that fits the whole environment. It is thus necessary to reason at the population level. This actually matches the intuition for the problem we want to solve: it is not possible to generate a single “perfect” test case to kill all mutants; on the contrary a global set of test cases has to be generated and improved to kill all mutants.

The bacteriologic algorithm for test cases generation takes an initial set of test cases as an input and produces a good set of test cases as an output. The quality of the set of test cases is evaluated by a *fitness function*. The test case generation algorithm evolves following an incremental process (each increment is called a generation) and consists of a series of mutations (using a *mutation function*) on test cases, to explore the scope of solutions. The final set is built incrementally by memorizing test cases that can improve the quality of the set. As the execution unfolds there are two test sets, the *solution set* that is being built, and the set of potential test cases, that we call a *bacteriologic medium*. Each generation is divided in four steps, and there can be several stopping criteria for the global process: after a number of generations, when a minimum fitness value is reached by the solution set, the fitness has not changed for a number of generations, etc The operations applied at each step are described in the following.

Let's call $\mathcal{TC}$ the input domain of the program.

Fitness function. fitness: $2^{\mathcal{TC}} \rightarrow \mathbb{R}^+$

The fitness function computes a real number that evaluates the quality of a set of test cases regarding the global objective. In the case of automatic test generation, this function can be based on the coverage rate of the control graph, mutation score or any other test adequacy criterion.

The memorization function (detailed below) requires the fitness of one test case. A function called *relFitness:* $\mathcal{TC} \times 2^{\mathcal{TC}} \rightarrow \mathbb{R}^+$ computes the fitness of a test case tc (relatively to the fitness of a set of test cases TCS) as follows: relFitness(TCS, tc) = fitness(TCS$\cup${tc})-fitness(TCS)

Memorization function. mem : $\mathcal{TC} \rightarrow$ boolean

This function takes a test case as an input and returns true if it can be memorized in the solution set. This function thus computes the relative fitness of the test case. If the fitness satisfies a given condition, the function returns true and the bacterium can be memorized. For example, one condition is: a test case can be memorized if its fitness is greater than a given threshold (*memorization threshold*).

Mutation function. mutate : $\mathcal{TC} \rightarrow \mathcal{TC}$

The mutation function generates a new test case by slightly altering an ancestor bacterium. This operator is crucial for the algorithm, since it is the one that actually creates new information in the process. We can note that by recursive applications of this operator we should explore the whole set of possible test cases $\mathcal{TC}$..

Filtering function. filter : $2^{TC} \rightarrow 2^{TC}$

This function aims at periodically deleting useless test cases from the bacteriologic medium to control the memory space during the execution.

Apart from these four functions, two parameters need to be fixed to run the algorithm:

1. the *memorization threshold* which is introduced in the memorization function to limit the number of memorized test cases.
2. the *size of the test cases*. If the grammar for test cases is available, the size is the number of nodes in the syntactic tree.

The algorithm manipulates test cases that all have the same size. This may appear as a limitation, but it is necessary, because if the mutation function can make test cases grow to improve their fitness, the size of memorized test cases will actually always grow. Indeed, a bigger test case is always more fitted than a smaller one (this seems obvious intuitively, and was experimentally verified). However, the bigger a test case is the longer it takes to execute. On the other hand, if test cases are too small they can not kill enough mutants, or they kill so few that we need a very large set of test cases to reach a good mutation score. It is thus important to have a fixed size for test cases that needs to be tuned before running the algorithm.

4.1. Running a bacteriologic algorithm

This section goes into practical details for the application of a bacteriologic algorithm. The discussions focus on the specific issue of mutation score optimization and are illustrated with the C# parser example.

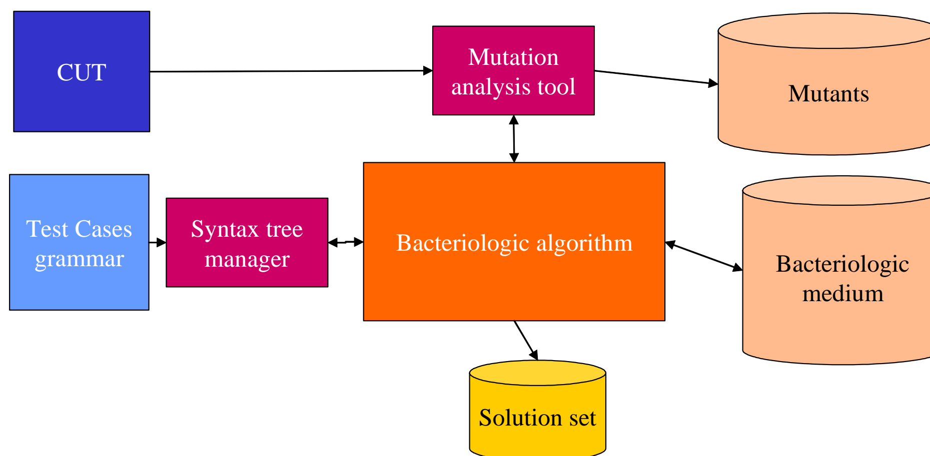

Figure 3 - Framework for test cases optimisation with a bacteriologic algorithm

Figure 3 displays the global architecture we used to automatically generate a set of test cases for the C# parser. However, this architecture is generic enough to be adapted to any problem that consists in improving the mutation score of test cases where structure can be described with a grammar. There are three main components: the bacteriologic algorithm, the mutation tool, and a syntactic tree manager. The process takes two input data: the component under test (CUT) and the grammar describing the test cases for the CUT. The output is a set of test cases with a high mutation score.

At the centre of the architecture is the bacteriologic algorithm component that handles the iterative process made of the four functions and that also manages two data sets: the bacteriologic medium and the solution set.

The mutation analysis tool is used to compute the fitness of test cases (since the mutation score of test cases is actually the function we want to maximize). More details on this part are provided in the subsection dedicated to the computation of the fitness function. The syntactic tree manager (STM) is specific to the case where the grammar for the test cases is available (in our example, the grammar is the C# grammar and test cases are C# programs). As it will be detailed below, the STM is used both for initialisation and for the mutation function of the bacteriologic algorithm.

a) Initialization

The initial set of test cases can either be written by hand or automatically generated with a random generator. In the architecture proposed in Figure 3, the initial set of test cases can be randomly generated by the STM when the grammar for test cases is available. For the experiment with the C# parser, the initial set of test cases was randomly generated from the C# grammar.

We conducted several experiments to tune the size of the test cases [11] and it was fixed to 25 (as the example of Figure 2). This size was passed as a parameter to the STM to generate initial test cases: C# programs which syntactic tree is made of 25 nodes.

b) Fitness function

When optimizing the mutation score of a set of test cases, the fitness function is the mutation score for a set. The computation of this function is thus based on a mutation tool. We developed a tool called NMutator, dedicated to the C# language, that is able to automatically generate all mutants for the NOR operator. This tool actually parses C#

components to find all possible locations in the code where it can introduce an error. Then it generates all corresponding mutant components.

Once all mutants are available, NMutator takes a set of test cases as an input and automatically executes all test cases against each mutant. For each test case, the tool saves the set of mutants it can kill. It can then compute the mutation score for each test case. Making the union of the sets of mutants killed by all the test cases, the tool can also compute the global mutation score for the set of test cases.

c) Memorization function

This function computes the relative fitness of all test cases in the bacteriologic medium. In our case, this is the mutation score of a test case relative to the mutation score of the solution set. This relative fitness thus computes the proportion of mutants a test case tc can kill that are not killed by the test cases that are in the solution set. If we call MS the mutation score of a set of test cases, relMS (the relative mutation score) is expressed as: $relMS(TCS,tc)=MS(TCS\cup\{tc\})-MS(TCS)$.

Since NMutator associates a set of killed mutants to all the test cases, it can easily compute $MS(TCS\cup\{tc\})$ by merging the sets of mutants killed by test cases in TCS and the set of mutants killed by tc.

Once the relative mutation scores have been computed for all test cases in the bacteriologic medium, the memorization function selects the ones that have a relative mutation score greater than the memorization threshold (which is a global parameter of the algorithm).

d) Mutation function

The mutation function randomly selects test cases in the bacteriologic medium. The random selection is weighted by the relative fitness of the test cases (the better test cases have higher chances to be selected for mutations). All selected test cases are then mutated to create new test cases that are added to bacteriologic medium for next generation. In the case of the C# parser, test cases are C# programs. The test cases can thus be seen as the syntactic tree representing the program, and mutating a test case then consists in replacing one node in the tree by another licit node. The STM is in charge of this mutation. Since it has access to the grammar of the test cases, it is able to parse a source test case, select a node in the tree and find a licit node to build a target test case (by licit node, we mean that the node replacement must build a syntactically correct test case).

As an example in the test case of Figure 2, the `foreach` node (lines 17 and 18) can be replaced by a while node like the one sown below:

while(cond1){aVariable1++;}}

e) Filtering function

Two different implementations of this function are used to delete test cases from the bacteriologic medium:

- Delete test cases which relative mutation score is equal to 0 (it kills no mutant that are not killed by the test cases in the solution set)
- Reduce the coverage matrix. This operation consists in deleting redundant test cases. For example, it is possible that some test cases kill the same mutants. It is useless to keep all of them.

Apart from these two functions, many other techniques for minimizing or prioritising test cases sets could be used. This problem has been studied in several works [12].

4.2. Results

Figure 4 shows results of a bacteriologic algorithm execution for the C# parser. The aim of the experiment was to generate a set of test cases that could kill the 500 mutants generated for the parser. The initial set of test cases was randomly generated and was composed of 30 test cases. The best one had a 57% mutation score and was memorized at the first generation (initial score of the solution set). The size of test cases was 25 nodes in the syntactic tree and the memorization threshold was 20%. After 30 generations, the algorithm generated 7 new test cases and the final set had a mutation score of 96%. We can notice that the generated test cases allowed us to actually detect errors in the parser. After fixing these errors, we ran the bacteriologic algorithm again (changing the component changes the set of mutants, and it is necessary to run another mutation analysis with these new mutants). With such an incremental process, we were able to establish a good confidence both in the set of test cases and the component.

Since the bacteriologic algorithm is a pseudo-random algorithm, the results, with the same set of mutants, slightly varied from one execution to the other. For example, the number of generated test cases varied between 7 and 10 from one experiment to the other.

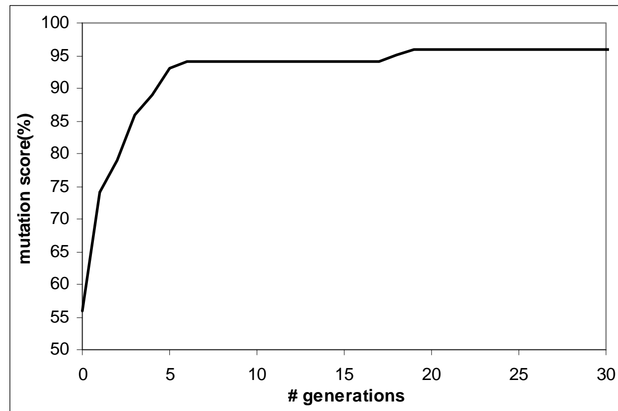

Figure 4 - Results of a bacteriologic approach for system test data optimization

Since genetic algorithms have often been used for automatic test cases generation, in [5] we compare the results of a bacteriologic algorithm with a genetic algorithm. Each algorithm has been excuted 50 times and the given mutation scores are average values. The results are summarized below:

- genetic algorithm: 200 generations for an average mutation score of 85% (ranging from 80% to 87%). Each run requires the execution of 480000 test cases.
- bacteriologic algorithm: 30 generations for an average mutation score of 96% (ranging from 92% to 97%). Each run required, in average, the execution of 46375 test cases.

Looking at the number of generations needed to reach the best score, it appears that a bacteriologic approach converges faster than a genetic one: 30 generations instead of 200. However, since the computation performed to go from one generation to the other is not the same in both approaches, we give more comparable figures: the number of times a mutant program has been executed. This is a better estimation of the complexity than the number of generations since executing a mutant is as much time-consuming in both approaches.

Another advantage of the bacteriologic algorithm is that it is easier to tune than the genetic one. This makes the bacteriologic approach more reusable for test generation/optimization problems. Removing parameters also makes the model more controllable since there is less randomness in the algorithm's execution. The approach is thus more stable than a genetic one.

5. Conclusion

The general purpose for this work is to asses the quality of test cases for a component, the intuition being that efficient test cases allow good confidence in the component that passes these test cases. The quality of test cases is evaluated by the mutation score, which

corresponds to the proportion of injected faults the test cases are able to detect. The work presented in this paper tackles the particular issue of automating the improvement of an initial set of test cases. Indeed, experiments show that, if it can be easy to write test cases that kill around 60%, improving this score up to 90% is very difficult and time-consuming.

We presented an original algorithm for automatic optimization of a set of test cases. This algorithm, inspired by the biological phenomenon of evolutionary ecology, is called *bacteriologic algorithm*. This optimization algorithm is used to generate a set of test cases that has a good mutation score. It takes a set of test cases as an input (that can be randomly generated or written by hand). Moreover, two parameters need to be fixed to run the algorithm: the size of the test cases (all test cases in the result set have the same size), and the memorization threshold. The algorithm then produces a set of test cases that has a good mutation score as an output. In this paper, we presented the algorithm with an example based on a parser for the C# language.

Acknowledgments.

The authors are grateful to Stan Rifkin for his very helpful comments on earlier version of this paper. Many thanks also to Françoise Burel, director of the “Ecosystem Functioning and Conservation Biology” lab of Rennes I University, for her helpful suggestions in the definition of the bacteriologic algorithm.

References

1. DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer.* IEEE Computer, 1978. **11**(4): p. 34 - 41.
2. Kim, S.-W., J.A. Clark, and J.A. McDermid, *Investigating the effectiveness of object-oriented testing strategies using the mutation method.* Software Testing, Verification and Reliability, 2001. **11**(4): p. 207 - 225.
3. Baudry, B., et al. *Trustable Components: Yet Another Mutation-Based Approach*. in *1st Symposium on Mutation Testing*. 2000. San Jose, CA.
4. Moore, I. *Jester - a JUnit test tester*. in *XP'2001*. 2001. Villasimius, Sardinia.
5. Baudry, B., et al. *Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment*. in *ISSRE'02 (Int. Symposium on Software Reliability Engineering)*. 2002. Annapolis, MD, USA.

6. Fleurey, F., *C# parser home page*. 2002.
7. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. 1995: Addison-Wesley.
8. Offutt, A.J., et al., *An experimental evaluation of data flow and mutation testing.* Software Practice and Experience, 1996. **26**(2).
9. Offutt, A.J., et al., *An Experimental Determination of Sufficient Mutant Operators.* ACM Transactions on Software Engineering and Methodology, 1996. **5**(2): p. 99 - 118.
10. Pianka, E.R., *Evolutionnary Ecology*. 1999: Addison-Wesley. 512.
11. Baudry, B., et al. *Automatic Test Cases Optimization using a Bacteriological Adaptation Model: Application to .NET Components*. in *ASE'02 (Automated Software Engineering)*. 2002. Edimburgh, Scotland, UK.
12. Rothermel, G., et al., *Prioritizing test cases for regression testing.* IEEE Transactions on Software Enginnering, 2001. **27**(10): p. 929-948.