Computational Intelligence for Testing .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, associated to a global testing-for-trust methodology, 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 automatically improving test cases efficiency. So, we looked first at genetic algorithms to solve the problem of test optimization and modeled it as follows: a test case can be considered as a predator while a mutant program (i.e. a program containing a fault) is analogous to a prey. The aim of the selection process is to generate test cases able to kill as many mutants as possible. Then, we propose a variation on this idea – and a new AI algorithm, no longer at the "animal" level (lions killing zebras) but at the bacteriological level. The bacteriological level indeed better reflects the test case optimization issue that is closer to an adaptation process that an evolution one. We describe these models and how the corresponding tools have been designed and implemented in C#. The approaches are compared on a .Net system.

1. Introduction

In an object-oriented (OO) development process, testing can no longer be separated from specification design and implementation stages. Most OO methodologies [1] insist on integrating design and test. 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* [2] to produce efficient test cases. The test cases can then detect weaknesses in the implementation and the specification. These weaknesses are fixed to make the three facets consistent with each other. The user can then iterate by running another mutation analysis with the improved implementations and specifications.

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 corresponds to a code generated from a faulty source code. We detail in the following how this issue can be overcame.

Mutation analysis has been successfully applied to qualify unit test cases for OO classes [3, 4], and gives the programmer an interesting feed-back on the "revealing power" of his/her test cases. It also offers an estimate of how much new test cases are needed to better test a given software component. While, the test cases provided by the tester generally kill 50-70 % of the mutants, improving this score up to 90-100 % is a time-consuming and a very expensive task.

The work presented in this paper focuses on automating the test cases improvement step: it will be also presented in [5]. This issue is a non-linear optimization problem, and the application of genetic algorithms (GAs) looks like an interesting way to solve it. Furthermore, a strong analogy exists between natural selection and the process of generating new test cases based on an initial set of test cases. Initial test cases are of various efficiency, but each of them can participate to the test optimization. In this paper we model the optimization problem as follows: a test case can be considered as a *predator* while a mutant program is analogous to a *prey*. The aim of the selection process is to generate test cases able to kill as many mutants as possible, starting from an initial set of predators, that is the test cases set provided by the tester. 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#

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

parser in the .Net framework. While it was quite disappointing to us that these experimentation results were not as good as we expected, we were suggested by biologist friends to try a slight variation on this idea, no longer at the "animal" level (lions killing zebras) but at the bacteriological level. The bacteriological level indeed better reflects the test case optimization issue: it mainly differs from the genetic one by the introduction of a memorization function and the suppression of the crossover operation. We describe this original bacteriological model and show how it behaves on the previous case study. The new results are very encouraging since the model converges faster than the first one, and is easier to tune and so, is more reusable.

Finally, to be systematic and as complete as we can in our study, we explored all the spectrum of possible algorithms, from genetic algorithms to bacteriological ones.

The rest of this paper is organized as follows. Section 2 opens with a brief summary about mutation analysis, and the methodology for trustable components. Section 3 presents a model for test optimization based on genetic algorithms. Section 4 presents the case study that has been conducted with this model, and discusses the results of these experiments. That leads to section 5 which presents an adaptation of the genetic model called the bacteriological model, new results are given.

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 [6]. It has been originally proposed in [2], and consists of creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test cases set that distinguishes the program from all its mutants.

The assumptions for mutation analysis are the competent programming hypothesis and the coupling effect. The competent programming hypothesis states that a programmer produces a program close to the correct one. Thus inserting minor errors in the program under test seems relevant to get closer to the correct program. The coupling effect states that if test cases can detect several programs with minor errors, they will detect more complex errors in the original program.

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*. 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.

A test cases set is *relatively adequate* if it distinguishes the original program from all its non-equivalent mutants. Otherwise, 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.

A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program test quality. During the test selection process, a mutant program is said to be *killed* if at least one test case detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test cases detect the injected fault.

2.2. Mutation analysis for .Net

Mutation analysis has only been applied mostly to validate unit test cases. In an object-oriented context, the class is often considered as the unit for testing, and mutation analysis has been successfully used to guide the generation of test cases for a class ([3, 4]). Testing components composed of interconnected classes is thus considered as system testing. However, it seems that only Olsson and Runeson have investigated mutation testing at system level in [7]. Generating system test cases for .Net components, based on mutation analysis, is a derived contribution of this paper.

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 [8], working at C# level will guarantee that faults have realistic semantics.

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

Among the complete list of mutation operators, we have chosen to select only a few of them, to avoid generating too many mutant components. These operators must generate only non-equivalent mutants. This subset of operators is still efficient since we expect classes to be tested at unit level (so all operators have been applied on the code separately). For this first study, we chose two mutation operators:

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

We use a specific oracle function based on behavioral difference. The oracle is obtained either by comparing the traces resulting from the outputs of the original with those of the mutants, or by comparing the object states of the original and mutant programs. The second comparison is useful for comparing programs that produce no analyzable outputs. In the studied .Net component, the first kind of oracle has been used. In the methodology presented in the following section, we claim that contracts should be used as oracles for improving global software trustability.

2.3. Testing for trust : a methodology

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). So, we present a testing for trust methodology particularly adapted to a design-by-contract approach [9]. The derivation may be automated from UML thanks to the OCL (Object Constraint Language) to a given language such as Eiffel# [10]. Considering language-independent contracts should be one of the expected features offered by the .Net paradigm (see [11] for a first practical contribution in this direction). As several testing frameworks now offer this feature to components under test [12], test suites are defined as being an "organic" part of OO software component: the component is made *self-testable* [13]. 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*). This view of an OO component is illustrated under the triangle representation (cf. Figure 1).

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 confrontation between these three facets leads to the improvement of each one. Before definitely embedding a test suite, the efficiency of test cases must be checked and estimated against implementation and specification, especially contracts. Tests are build from the specification of the component; they are a reflection of its precision. They are composed of two independent conceptual parts: test cases and oracles. Test cases execute the functions of the component. Embedded oracles – predicates for the fault detection verdict – can either be provided by assertions included into the test cases or by executable contracts. In a design-by-contract approach, our experience is that *most* of the verdicts are provided by contracts derived from the specification[4]. The fact that components' contracts are inefficient to detect a fault exercised by the test cases reveals a lack of precision in the specification. The specification should be refined and new contracts added. The trust in the component is thus related to the test cases efficiency and the contracts "completeness". We can trust the implementation since we have tested it with a good test cases set, and we trust the specification because it is precise enough to derive efficient contracts as oracle functions.

The question is thus to be able to measure this consistency. This quality estimate quantifies the trust one can have in a component: it is not, in the strict sense, a measurement, but an estimate that is in conformance with intuition. The chosen quality criteria proposed here is the proportion of injected faults the self-test detects when faults are systematically injected into the component implementation, i.e. the mutation score.

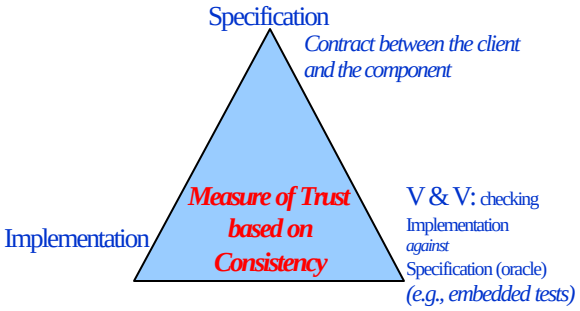

Figure 1 - Trust based on triangle consistency

The global component design-for-trust process consists of 6 steps that are presented in Figure 2.

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 used oracle function is the comparison between the testing object states/or output traces differences.
3. the user has to check if the tests do not detect errors in the initial program. If errors are found, he must debug them.
4. measure the contracts quality thanks to mutation testing. We use the embedded contracts as an oracle function here.
5. improvement of contracts to reach an expected quality
6. a global oracle function is build. To do this, it executes all the tests on the initial class, and the object's state after execution is the oracle value.

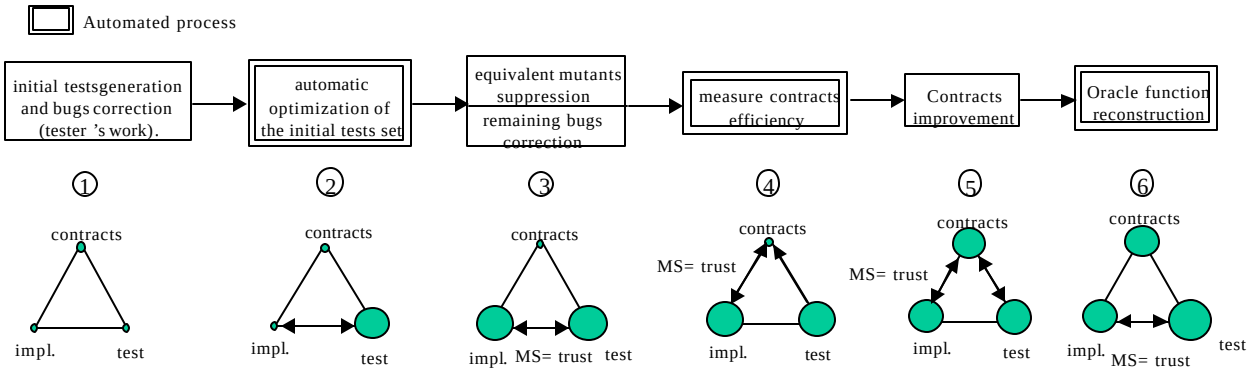

Figure 2 - The global testing-for-trust process

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 algorithms).

It has to be noticed that this stage does not depend on the contracts of the component: the proposed algorithms can be applied to the test optimization of a component without contracts.

3. Test cases generation : Genetic algorithms for test generation

In this paper, we argue that writing a first set of test cases is easy, and most developers do such basic testing. Our experiments showed that such test cases easily reach 60 % of test quality [4], whereas improving this score up to 90% implies a particular and specific supplementary testing effort. In this section we investigate the use of genetic 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.

3.1. Genetic algorithms for test optimization

Genetic algorithms [14] have been first developed by John Holland, whose goal was to rigorously explain natural systems and then design artificial systems based on natural mechanisms. So, genetic algorithms are optimization algorithms based on natural genetics and selection mechanisms.

To apply genetic algorithms to a particular problem, it has to be decomposed into atomic units that correspond to genes. In our case a *gene* corresponds to a *test case* for the component under test. The gene model is dependent of the case study: a C# parser. For this particular system, the input data is a source file that is parsed to build a syntactic tree. The gene model is given in the following definition. Then individuals can be build, corresponding to a finite string of genes, and a set of individuals is called a population. A second criterion needs to be defined : a *fitness function* F which, for every individual among a population, gives a fitness value F(x), which qualifies the individual regarding the problem we want to solve. This corresponds to the function we want to maximize.

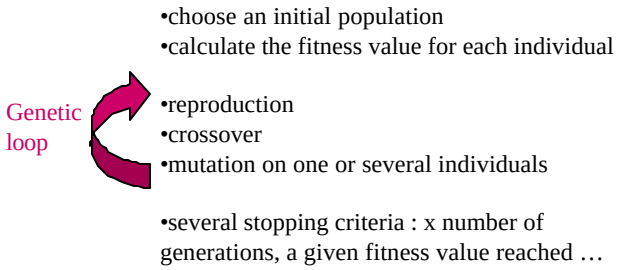

Figure 3 - The global process of a genetic algorithm

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.

- **Reproduction.** This operator builds a new population by copying the best individuals based on their fitness value, here the best mutation score.
- **Crossover** : let m be the size of an individual, and let's select an integer i at random between 1 and m-1, then from two individuals ind_1 and ind_2, we can create two new individuals ind_3 and ind_4 as follows :

$ind_1=\{G_{11}, \ldots G_{1i}, G_{1i+1}, .. G_{1m}\}$ $\quad ind_2=\{G_{21}, \ldots G_{2i}, G_{2i+1}, .. G_{2m}\}$

$\Downarrow$

$ind_3=\{G_{11}, \ldots G_{1i}, G_{2i+1}, .. G_{2m}\}$ $\quad ind_2=\{G_{21}, \ldots G_{2i}, G_{1i+1}, .. G_{1m}\}$

- **Mutation.** The mutation operator modifies one or several genes' value. (e.g. if an individual is a bit string, mutation means changing a 1 to 0 and vice versa). 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 as illustrated in the following figure:

$G=[N_1,\ldots, N_i,\ldots, N_x] \Rightarrow G_{mut} = [N_1,\ldots, N_{imut},\ldots, N_x]$

Then, the genetic algorithm is computed following the process described Figure 3.

Next section presents results of the application of a genetic algorithm to automatically improve the quality of test cases for a parser of the C# language.

4. Case study with genetic algorithms

This section describes a case study: 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 for the genetic algorithm application consisted of 12 individuals of size 4, and its initial mutation score was 56%. The results are given Figure 4.

4.1. Results and comments

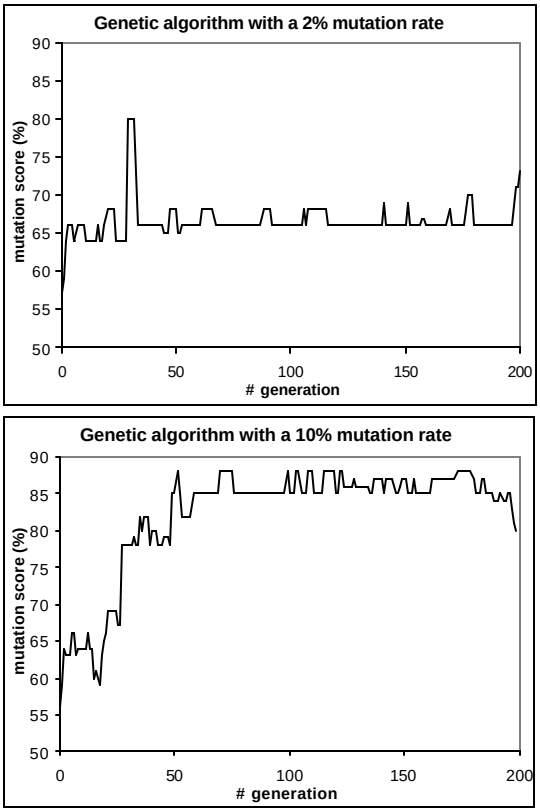

Figure 4 - Genetic algorithm application for test optimization for a C# parser

The experiments with genetic algorithms were not satisfactory. Both because of the slow convergence and the unusual proportions of crossover and mutation operators. We had to excessively increase the mutation rate compared to usual application of genetic algorithms (mutation rates lower than 2%). Figure 4 shows results with two different mutation rates: 2% and 10%.

As a conclusion about the case study, we can say that GAs are not perfectly adapted to the test cases optimization problem. A more adapted model should provide memory and remove the notion of individual to concentrate on the genes (test cases). Nevertheless, things must be kept from this experience.

- The notion of gene corresponds exactly to what has to be optimized.
- The mutation operator that seems to be a good way of creating new information to solve our problem.
- The mutation score as the fitness function that guides the algorithm towards a good solution.

Next section proposes a new model and process, adapted from the genetic algorithms and based on these conclusions. It is called the bacteriological approach, and is based on the bacteriological adaptation phenomenon.

5. An adaptive approach: Bacteriological algorithms

Experiments described in section 4 have shown some drawbacks of genetic algorithms for test cases optimization. Our new approach is thus fairly far from the genetic model. If we keep the analogy with biological processes, this new model is close to the "bacteriological adaptation" [15].

5.1. The bacteriological model

The bacteriological approach is more an adaptive approach than an optimization approach as was the previous one based on genetic algorithms. It aims at mutating the initial population to adapt it to a particular environment. The adaptation is only based on small changes on the individuals. The individuals in the population are called *bacteria* and correspond to *atomic units*. Unlike the genetic model the bacteria cannot be divided. Bacteria can only be reproduced and altered to improve the population.

As for the genetic model, a *fitness function* is necessary to choose bacteria for reproduction. With this function we can draw a global iterative process to adapt an initial population (Figure 5). Starting from this population, the fitness function allows the algorithm to select the best bacteria. Then these bacteria are saved and reproduced to generate a new population. Several bacteria in this population are mutated, then the best ones are selected again to produce another generation. This process stops after a number of generation or when the memorized population has reached a optimum fitness value.

Since the bacterium model is the same as the gene model, the fitness function is the same here as in the previous model. Bacteria are selected according to their mutation score.

In the genetic approach, the algorithm computed the mutation score of individuals on every mutants at each generation. Conversely the bacteriological approach aims

at avoiding this expensive mutation score computation by saving the best bacteria from one generation to the next. New data structures appear to manage this mechanism. First, a memory that contains the best bacteria. Second two sets of mutants: a set of dead mutants and a set of mutants still alive. These two sets are a partition of the global set of mutants for the component under test. A mutant is inserted in the set of killed mutants if it is killed by one of the memorized bacteria. With these structures available it is possible to compute the mutation only on mutants that have not been killed in previous generations.

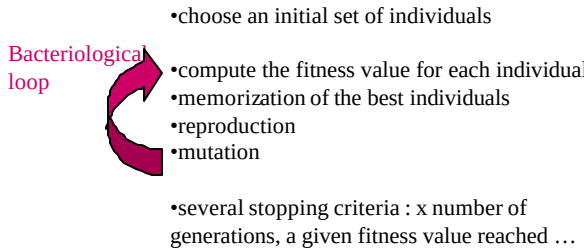

Figure 5 - The bacteriological process

At last, since this approach only manipulates bacteria which correspond to genes in the previous approach, the notion of individual disappears. The reproduction and crossover operators have thus also disappeared. The removal of the crossover operation is one major difference with the genetic model. This corresponds to an evolution we thought was necessary when looking at the result of genetic algorithms, since this operator did not help the convergence towards the optimal solution (see discussion in section 4.1).

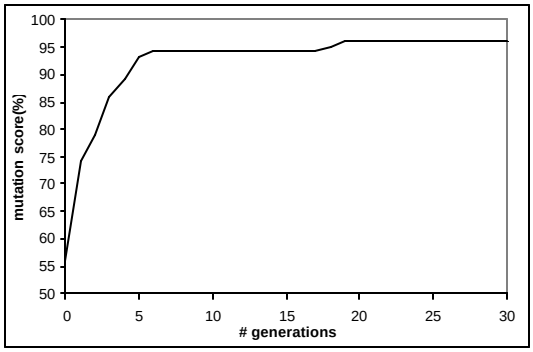

Figure 6 - Results of a bacteriological approach for test data optimization

5.2. New results

Figure 6 shows results of our bacteriological approach for the two case studies presented in section 4. For this type of experiment, only two parameters need to be tuned: the number of bacteria saved to pass from one generation to the other, and the minimal size of the bacteria. Since the initial bacteria pool was small (between 3 and 10 bacteria), the experiments were conducted by saving only the best bacterium for a given generation. The size of a bacterium is defined as the number of nodes of this bacterium.

Table 1 summarizes results of both approaches for the C# parser. This table gives the number of generations needed to reach the score given in the second column. The bacteriological algorithm converges much faster than the genetic one: 30 generations instead of 200. We also show how many times a mutant has been executed, which is a good estimate of the computation complexity.

As a conclusion about the new experiments, it seems that the bacteriological comparison leads to a better heuristic for our problem.

Table 1 - Comparison between genetic and bacteriological algorithms for the C# parser

Algorithm	# generation	mutation score (%)	# mutants executed
Genetic	200	85	480000
Bacteriologic	30	96	46375

6. Conclusion

The general assumption for this work is to measure the quality of test cases (the revealing power of the test cases [6]), 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, GAs and BAs (bacteriological algorithms).

The qualification of test cases based on mutation analysis has been adapted at component/system level. We experimented two different models derived from analogy with natural processes, for test optimization. Genetic algorithms were studied first. Because of the disapointing results, we introduced an original model, to simulate the bacteriological adaptation phenomenon. 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). Intermediate algorithms have also been studied into the spectrum of possible algorithms between ‘genetic’ and ‘bacteriological’ algorithms.

References

[1] P. Kruchten, "The Rational Unified Process, An Introduction". Object Technology Series, Addison-Wesley, 2000.

[2] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection : Help For The Practicing Programmer". IEEE Computer. Vol.11(4), p. 34-41, 1978.

[3] S.-W. Kim, J.A. Clark, and J.A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method". Software Testing, Verification and Reliability. Vol.11(4), p. 207-225, 2001.

[4] B. Baudry, Y. Le Traon, J.-M. Jézéquel, and V.L. Hanh. "Trustable Components: Yet Another Mutation-Based Approach". in proceedings of *1st Symposium on Mutation Testing*, San Jose, CA, October 2000.

[5] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment". in proceedings of *ISSRE 2002*, Annapolis, MD, USA, November 2002.

[6] J.M. Voas and K. Miller, "The Revealing Power of a Test Case". Software Testing, Verification and Reliability. Vol.2(1), p. 25-42, 1992.

[7] T. Olsson and P. Runeson. "System Level Mutation Analysis Applied to a State-Based Language". in proceedings of *International Conference and Workshop on the engineering of Computer Based Systems (ECBS'01)*2001.

[8] Salamander. "Salamander, a .NET decompiler" http://www.remotesoft.com/salamander

[9] B. Meyer, "Applying Design by Contract". IEEE Computer. Vol.25(10), p. 40-51, 1992.

[10] R. Simon, E. Stapf, and B. Meyer. "Full Eiffel on the .NET Framework" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp

[11] K. McFarlane. "Design by Contract Framework" http://www.codeproject.com/csharp/designbycontract.asp

[12] P. Craig. "NUnit" http://nunit.sourceforge.net/

[13] J.-M. Jézéquel, D. Deveaux, and Y. Le Traon, "Reliable Objects: a Lightweight Approach Applied to Java". IEEE Software. Vol.18(4), p. 76-83, 2001.

[14] D.E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley, 1989.

[15] M.L. Rosenzweig, "Species Diversity In Space and Time", Cambridge University Press, 1995.