Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment

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

The level of confidence in a software component is often linked to the quality of its test cases. This quality can in turn be evaluated with mutation analysis: faulty components (mutants) are systematically generated to check the proportion of mutants detected ("killed") by the test cases. But while the generation of basic test cases set is easy, improving its quality may require prohibitive effort. This paper focuses on the issue of automating the test optimization. We looked at genetic algorithms to solve this problem and modeled it 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. To overcome disappointing experimentation results on the studied .Net system, we propose 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 introduces of a memorization function and the suppresses the crossover operator. We describe this model and show how it behaves on the case study.

1. Introduction

Some specialists have claimed: "Programmers love writing tests" [1]. One reason for this is that they can incrementally build confidence in their code when it passes their tests. The level of confidence one has into a given software component is then linked to the quality of its test cases. Conversely, one way to qualify the test cases consists in deliberately introducing faults in the software under test. The intuition of this technique, called *mutation analysis* [2], is that the quality of the test cases is related to the proportion of faulty programs (also called *mutants*) it detects. Faulty programs are generated by systematic fault injection in the original implementation. By measuring the quality of test cases (the revealing power of the test cases [3]), we seek to build trust in a component passing those test cases. Mutation analysis has been successfully applied to qualify unit test cases for OO classes [4-6], and gives the programmer an 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.

But while the generation of a set of basic test cases is easy, improving its quality may require prohibitive effort. Indeed, the test cases that are generally provided by the tester easily cover 50-70 % of the mutants, but improving this score up to 90-100 % is a time-consuming and a very 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, 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# parser in the .Net framework [7, 8]. 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.

The rest of this paper is organized as follows. Section 2 opens with a brief summary about mutation analysis, and introduces how it is adapted to test generation and optimization. A derived contribution of this paper concerns the adaptation of the mutation approach to a whole system. Section 3 presents a model for test optimization that builds 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. In section 6 some related work are discussed and section 7 gives several conclusions about this work.

2. Mutation testing for OO domain

Mutation testing is a technique which was first designed to create effective test data, with an important fault revealing power [3, 9]. It has been originally proposed in 1978 [2], and consists in creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a set of test cases that distinguishes 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. To create a mutant, it is sufficient to apply its associated operator to 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.

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

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. We recall the definition of the mutation score:

In the following, we will first detail which mutation operators were used for experiments. This choice has been guided by the specific use of mutation analysis for test cases at system level. Then, we describe the general test selection process based on mutation analysis, and pinpoint which part of the process we want to automate.

2.1. Mutation analysis for system testing

Mutation testing has mostly been applied at unit level. 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 ([4-6]).

When applying mutation analysis for system testing, scale problems appear. In the following, we call a *mutant program*, a software system in which an error has been injected. A system is composed of several classes, and each of them can generate many mutants (many faults can be injected). For example, in [4, 5], a large number of operators is used which generate large sets of mutants that are necessary to have a precise evaluation of test cases for one class. The number of mutant programs thus increases with the size of the system under test. Moreover, since all the test cases must be executed against all the mutants, the execution time increases with the number of mutants. Mutation analysis at system level can thus become very time-consuming. At last, if mutant equivalence is often decidable on a class, it is not possible for a tester to decide system equivalence.

The solution we have chosen is to select two mutation operators to avoid generating too much mutant programs. This subset of operators is still efficient since we expect classes to be tested at unitary level (so all operators have been applied on the code separately). System testing then focuses on the relationships between the classes in the system. Since the purpose of unit and system testing is different, mutation analysis also has to have a different role. 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.

Other solutions exist to avoid execution time expense. For example, we could have selected the classes to be mutated in the system in function of a criterion such as the "distance" between interface classes and classes under test. The strategy would inject faults only in classes that are difficult to control from the system interfaces. Another solution would consist in finding mutation operators more specific to system testing. We could think about errors on the UML model of the system under test, as Olsson and Runeson did on SDL models in [10].
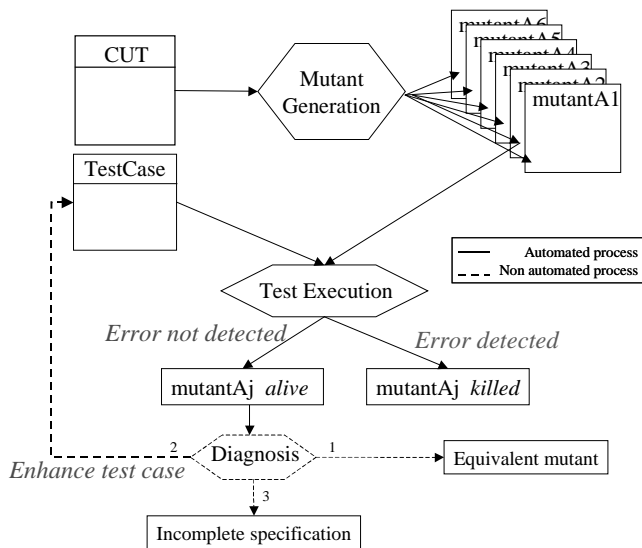
2.2. Test selection process

Figure 1 - The mutation process

The whole process for generating test cases with fault injection is presented in Figure 1. First, a set of mutant programs is automatically generated with the selected mutation operators. Then, the test case is ran against each mutant. An oracle function is used to determine if the test case has killed the mutant. This oracle is specific to the mutation analysis, and is based on the assumption that the original program is correct. It consists in comparing the behavior of the original program and the behavior of the mutant program. Let Out_o be the set of outputs when running a test case with the original program, and Out_m be the set of outputs for the test case with a mutant program. If $Out_m \neq Out_o$, then the mutant is killed by the tests set.

If a mutant program is not killed by any test case, the diagnosis step determines the reason of non detection. The mutant may be alive because of a test case too weak, because specification is incomplete, or because it is an equivalent mutant. This diagnosis step is the only one in the process that is not automated.

In this paper, we focus on the automation of the test case enhancement phase after the diagnosis step. That is, we focus on the test generation process to automatically obtain the most efficient set of test cases both in terms of fault revealing power (measured using mutation) and execution time (this aspect being crucial for testing a system). In Figure 2 an "optimizer" operation has appeared that optimizes the initial test case to improve its mutation score. As it will described in the following sections, we have tried different strategies to automate the "optimizer" operation : genetic algorithms (section 3) or an adaptation of these algorithms that we have called bacteriological algorithms (section 5).
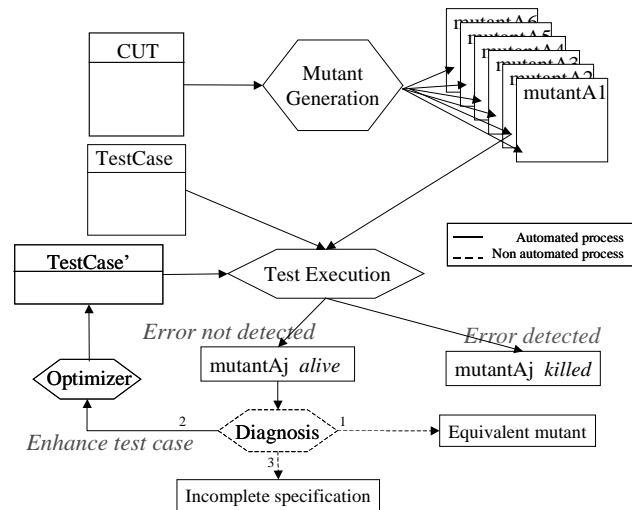

Figure 2 - Automation the test enhancement step in the mutation process

In [5], we had proposed a testing-for-trust methodology based on an integrated design and test approach for OO software components, particularly adapted to a design-by-contract approach, where the specification is systematically transformed to executable assertions (invariant properties, pre/postconditions of methods) [11]. Here we focus on test generation/optimization and we can extract the corresponding stages from the global methodology. Based on the process of Figure 2, Figure 3 proposes an incremental approach for testing and correcting software:
1. Write an initial test cases set
2. Automatically enhance the initial test cases set.
3. The tester checks if the tests do not detect errors in the initial program. If errors are found, they must be corrected. then go back to step 2 for regression testing.
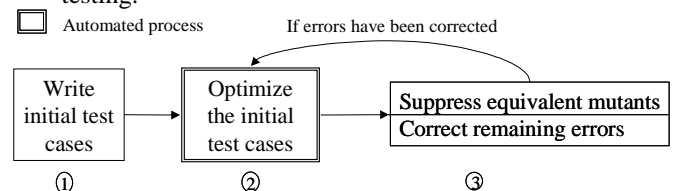

Figure 3 - Incremental process for software testing

Figure 4 displays a UML class diagram for the global architecture to apply our test generation methodology. There is a central `TestRunner` class that manages the relationship between the `ComponentUnderTest` (CUT) and the mutation tool represented (`Mutator` class), or the `TestOptimizer`. In this paper we are interested in a particular type CUT : C# components. A CUT has a set of associated test cases represented as an association between `ComponentUnderTest` and `TestCase`.
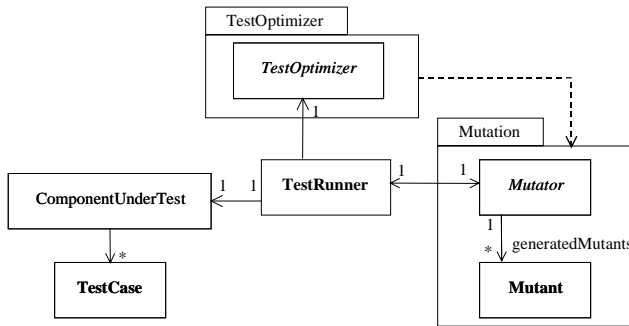
Figure 4 - Global architecture for test generation

This section detailed the mutation analysis and how it is inserted in a global test generation process. Figure 2 shows the part of the process on which this paper focuses: the automatic improvement of a test cases set for a component under test. Next sections present two models we have experimented for this purpose.

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 (see [12]). Improving test quality 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. Indeed, the basic test cases set carries information that can be optimized to create better test cases, by some cross-checking and "mutation" of the test cases themselves. So, at the beginning we have a population of mutants programs to be killed and a test cases pool. We randomly combine those test cases (or "gene pool") to build an initial population of test cases seen as predators of the mutant population. From this initial population, we apply a genetic algorithm to improve its ability to kill mutants programs.

3.1. Genetic algorithms

Genetic algorithms [13] have been first developed by John Holland [14], 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. In nature, creatures which best fit their environment (which are able to avoid predators, which can handle cold weather…) reproduce and thanks to crossover and mutation, the next generation will fit better. This is just how a genetic algorithm works: it uses an objective criteria to select the fittest individuals in one

population, it copies them and creates new individuals with pieces of the old ones.

This objective criteria used to go from one generation to the other is one of the interesting points of genetic algorithms, but there are others. As we will see, these algorithms are computationally simple, they improve rapidly and they work at the population level, not on a single individual.

To apply genetic algorithms to a particular problem, it has to be decomposed in atomic units that correspond to genes. Then individuals can be build, corresponding to a finite string of genes, and a set of individuals is called a population. All the individuals in a given population have the same size (the same number of genes). A second criterion needs to be defined : a *fitness function* F which, for every individual among a population, gives F(x), the value which is the quality of the individual regarding the problem we want to solve. This corresponds to the function we want to maximize.
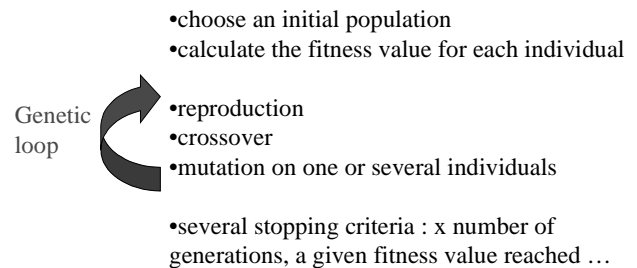

Figure 5 - The global process of a genetic algorithm

Moreover, a genetic algorithm uses three operators: reproduction, crossover, mutation.

- **Reproduction.** This operator copies the individuals which are going to participate in crossover: they are chosen according to their F(x) value. The choice can be seen as spinning a roulette wheel where each individual has a slot proportional to its fitness value. We spin the wheel as many times as the size of the population, and so we have a new population which is going to participate to crossover. This new population is made of individuals of the old one, and the number of each type of individual is proportional to its fitness (there are many of the fittest and few of the ones with a low fitness).

- **Crossover.** The members of the population after reproduction are mated randomly, then every pair is crossed, to create as many new pairs, like this : first, you choose, at random, an integer value k between 0 and the size n of an individual less one. Secondly, you create two new individuals A' and B' with a pair (A,B), A' is made of the k first genes of A and n-k last genes of B, and B' is made of the k first genes of B and the n-k last genes of A.

- **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)

Once the problem is defined in terms of genes, and the fitness function is available, a genetic algorithm is computed following the process described Figure 5.

Next section presents a model to apply genetic algorithms for automatic optimization of an initial tests set. We present how genes are modeled for this particular problem, as well as the three operators and the fitness function.

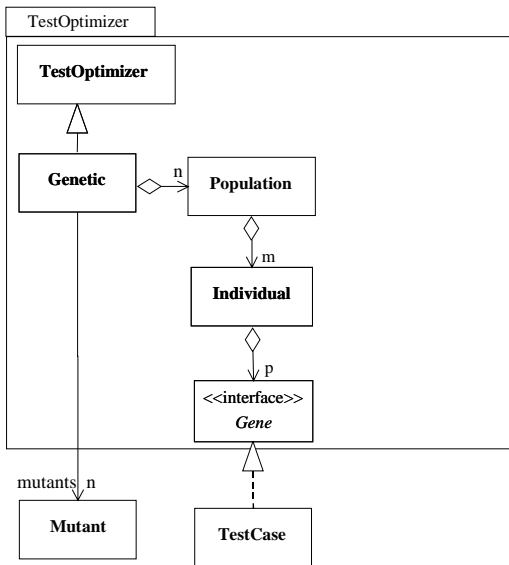## 3.2. Genetic algorithms for test optimization

Figure 6 - Architecture for test optimization using a genetic algorithm

Figure 6 presents a global architecture for test optimization using a genetic algorithm. Genetic algorithms present one way to automate test optimization, the `Genetic` class is thus a specialization of `TestOptimizer`. The decomposition of the problem as presented in section 3.1 appears clearly: a population is a set of individuals, and an individual is a set of genes. The size of the population and the size of an individual are constant values for a given run of the genetic algorithm.

The *gene modeling* is specific to the application of genetic algorithms to a test optimization problem. In this case a gene corresponds to a *test case* for the component under test. This appears on the architecture as a `TestCase` class that implements the `Gene` interface. The gene model is strongly dependent of our case study: a 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.

Gene modeling for test optimization. *For the problem of test optimization, a gene is modeled as a test case. In the particular case of a parser a gene is a source file for the particular language. Each file contains several constructs from the language (nodes from the syntactic tree). If there are x nodes in the file a gene can be represented as follows:*

$$G = [N_1,...,N_x]$$

Another aspect of the genetic algorithm has to be decided for the particular problem of test optimization: the *fitness function*. We have chosen the *mutation score* of an individual as the fitness function. The `Genetic` class encapsulates a set of mutants that is used to compute the fitness function for every gene.

Fitness function. *The fitness value for an individual is its associated* mutation score. *An individual is a set of genes. Let $I=[G_1, ..., G_n]$ be an individual composed of n genes. Let S_i be the set of mutants detected by G_i. At last, let nbMutants be the total number of mutants generated for the component under test. The fitness function of individual I is computed as follows:*

$$F(I)=(\frac{card(\bigcup_{i=1}^{n}S_i)}{nbMutants})\times100$$

The union set of all S_i corresponds to the set of mutants killed by the individual. The cardinal of this union is thus the number of mutants killed by the individual. Then the mutation score of the individual is the percentage of the global set of mutants it can kill.

Now, let us define the genetic operators for the particular problem of test cases optimization.

- **Reproduction** : the slot for each individual in the roulette wheel, is proportional to its mutation score.
- **Crossover** : let m be the size of individuals in a population, 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; one made of the i first genes of ind_1 and the m-i last genes of ind_2, and the other made of the i first genes of ind_2 and m-i last genes of ind_1. This operator in the following figure.

$ind_1=\{G_{11}, ... G_{1i}, G_{1i+1}, .. G_{1m}\}$ $ind_2=\{G_{21}, ... G_{2i}, G_{2i+1}, .. G_{2m}\}$
$$\Downarrow$$
$ind_3=\{G_{11}, ... G_{1i}, G_{2i+1}, .. G_{2m}\}$ $ind_2=\{G_{21}, ... G_{2i}, G_{1i+1}, .. G_{1m}\}$

- **Mutation**. Based on the gene modeling, the mutation operator consists in replacing a syntactic node in a source file (an individual) by another licit node. The class hierarchy for the node types makes it easy to build a compatible node, once the node to be mutated has been chosen (cf. Figure 7). 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:

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

Concrete examples of a source file, and the how it is mutated in are given in appendix A.

The problem when mutating one gene is to generate a new test case which is syntactically correct. For our case study (a parser for the C# language), this is made easy thanks to the particular structure of the test cases. Mutating a gene consists in replacing a node from the syntactic tree by another one. Since these nodes are hierarchically ordered (see Figure 7), a node must be replaced by a node which is at the same level in the tree (a brother node) to build a new correct test case. For example, a method can be replaced by either a destructor, a constructor, a field or a property.

Based on this model of the test case optimization problem , next section proposes an experiment using a genetic algorithm. It gives results of the application of a genetic algorithm to automatically improve the quality of test cases for a parser for the C# language.

4.   Case study with genetic algorithms

This section describes a case study that has been conducted to investigate the automation of test cases optimization using a genetic algorithm. It applies a genetic algorithm to optimize tests for a small system written in C# in the .NET framework. This system implements a simplified parser for the C# language. 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.

4.1. Test data optimization : testing a .NET component

To apply our test data optimization technique for system testing, we chose as a case study a .Net component that parses C# source files [8]. The UML class diagram for this parser is given Figure 7. There are 32 classes in this system that can be divided in three main parts. First, the CSNodeBuilder class which is the main class for building the syntactic tree. Second the inheritance hierarcky under the CSNode corresponds to the different types of nodes that can appear in the syntactic tree of a C# program. The third part of the diagram is the NodeVisitor interface and its different implementations. These classes correspond to the implementation of the Visitor design pattern [15], which enables to implement different treatments on the syntactic tree. For example, the TextCSPrettyPrinter class implements a textual pretty printer for the tree. This parser has been implemented in C#. This parser takes a set of C# source files as an input and builds the corresponding syntactic tree. To experiment genetic algorithms on this system, we generated 500 mutant programs, using only the NOR operator, we did not have time to implement the LOR operator (cf. section 2.1). Nevertheless the obtained results are still interesting since the test cases generated against such mutants cover all statements in the system. Most of the mutants were created from the classes TextCSPrettyPrinter, Tokenizer and CSNodeBuilder which process the most complex operations in the system. 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 8.
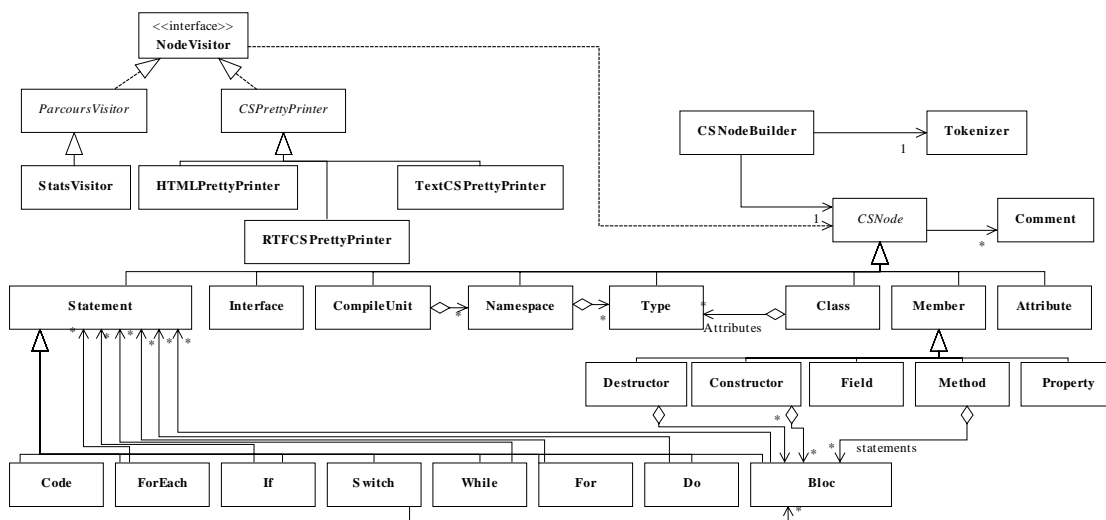

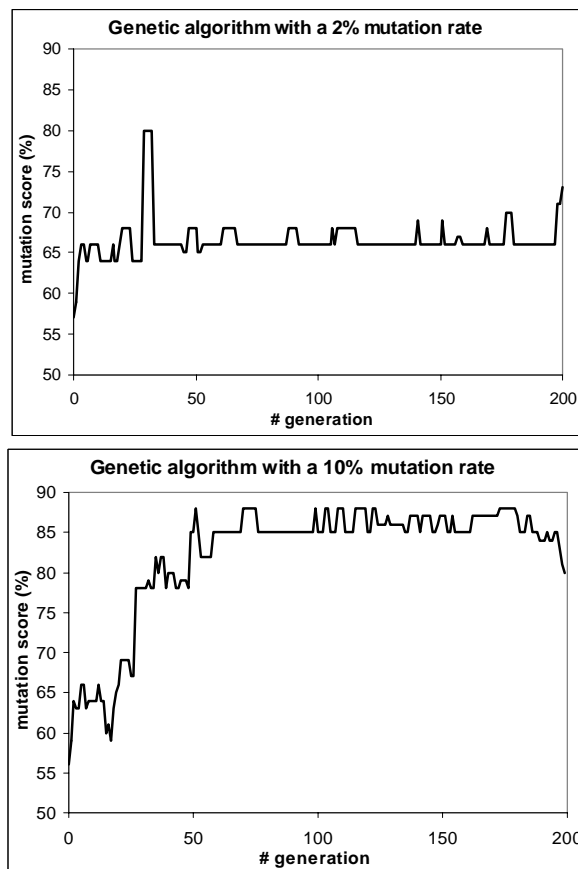**Figure 7 - Parser for the C# language**

Figure 8 - Genetic algorithm application for test optimization for a C# parser

4.2. Results and comments

This section summarizes several conclusions about the application of a genetic algorithm to improve the quality of test cases (Figure 8). First we focus on the benefits of this approach, and how it helped improve the quality of the CUT. Then, we explain the lack of efficiency of this type of model for our particular problem. We tackle different points and difficulties encountered when using our genetic model: the irregular and slow growth of the mutation score, the cost in terms of execution time, and the problems of calibrating the model.

The genetic algorithm actually automatically improved the mutation score of the initial set of test cases. The optimized test cases were ran on the CUT (step 3 of process Figure 3). Several errors were found and corrected. We also studied alive mutants. We detected some equivalent mutants, and dead code. Some mutants were obviously not equivalent, but still alive, and they actually corresponded to errors that had been injected in dead code. However the experiments with genetic algorithms were not satisfactory. Both because of the slow convergence and the unusual proportions of crossover and

mutation operators. In the following, we draw conclusions about these results and what is needed for a model more adapted to test cases optimization.

To go from one generation to another, genetic algorithms select the best individuals. These individuals are then reproduced, crossed, and some of them are mutated. This gives a new population. Information may be present only in genes of individuals that have not been selected for reproduction. In the same way, mutating a gene may delete information. There may thus be some information loss when passing from one generation to the other. In that case, the best individual of the new population may be worst than the best one of the previous generation. This phenomenon implies a slow convergence, or even troughs in the population evolution. Memorizing the individuals before reproduction would solve this problem.

The second limitation of this approach is the tuning of the model. First, the size of an individual has to be decided. Genetic algorithms look for an optimal individual, not an optimal population. The individuals must thus be big enough, from the beginning, to contain enough genes (test cases) to reach the best fitness value (mutation score). It is very difficult to predict how many test cases will be necessary to kill every mutants for a particular test cases set. So, we have to start with big sizes, then tune this parameter, so that the final individual (test cases set) has a good mutation score but is not too big. Big sets are not interesting because running all the test cases is too much time-consuming. The tuning has to be done for every particular CUT. Even if this tuning is mandatory when applying genetic algorithms for a particular problem, it seems particularly constraining in our case since our objective is to improve test cases and not test cases set. Our goal, is to have a set of good test cases, and not a good test cases set. Thus, we would need a model, that does not constrain the size of the set when improving the test cases.

The second important parameter that has to be tuned is the mutation rate. We had to excessively increase the mutation rate compared to usual application of genetic algorithms. Figure 8 shows results with two different mutation rates: 2% and 10%. For the lowest rate, the mutation score reaches at most 80%, whereas the 10% rate makes the mutation score grow up to almost 90%. Actually, it appears that the mutation operation, when running a genetic algorithm, is the one that creates information since it is the only operation that modifies the test data. So after mutation, the test case might cover other parts of the CUT. For test optimization, this represents an information saving.

At last, let's look at the crossover operator. The limitation of this operator is not so much the tuning, but the lack of efficiency. Indeed, the way genes are modeled as test cases implies that each gene can be run on the CUT separately. The genes are thus independent from each

other. So the order in which they are run as no importance. This makes the crossover operator useless, since its only function is to create information by reordering genes inside an individual.

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). This would avoid some tuning when applying the model on different CUTs. Nevertheless, things must be kept from this experience. The gene modeling which is clearly defined corresponds exactly to what has to be optimized. The mutation operation seems to be a good way of creating new information to solve our problem. The mutation score as the fitness function 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 the problem of test cases optimization. This section presents a specialization of the genetic approach for this particular problem. The adaptation consists in keeping track of the best individuals from one generation to the other. It is then possible to delete the mutants those individuals can kill from the set of alive mutants. The time necessary to compute one generation then decreases at each step of the genetic loop with the size of the alive mutants set.

Even if the adaptation of the genetic model seems based on very small changes, it actually completely changes the idea of genetic algorithm which is to go through the set of solutions looking for the optimal individual. Here, the set of solutions changes from one generation to the other since the goal of the search (killing every alive mutant) changes at each generation. Moreover, our new model does not generate the optimal individual, but a set of individuals (the ones that have been memorized during the whole process). This 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 "bacteriologic adaptation" [16].

5.1. The bacteriological model

The bacteriological approach is more an adaptive approach than an optimization approach as 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 can not be divided. The crossover operation can not be used anymore. Bacteria can only be reproduced and altered to improve the population.

As 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 9). 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 an optimum fitness value.
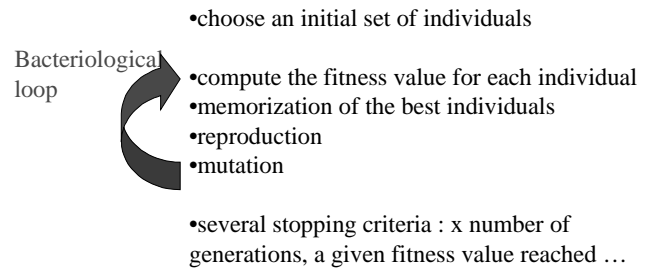

•choose an initial set of individuals

•compute the fitness value for each individual
•memorization of the best individuals
•reproduction
•mutation

•several stopping criteria : x number of generations, a given fitness value reached …

Figure 9 - The bacteriological process

5.2. The model for test optimization

Figure 10 displays a UML class diagram for the new model. The bacteriological approach is another technique for test optimization, thus it specializes the `TestOptimizer` class. A *bacterium* is modeled as a *test case* which structure is given in section 3.2.
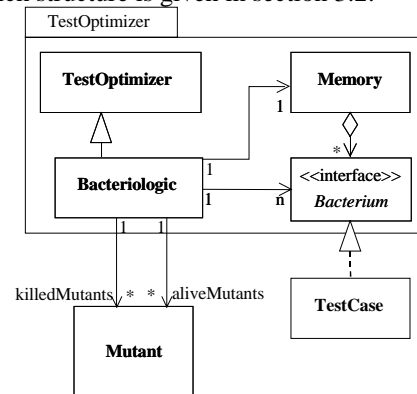
Figure 10 - Architecture for bacteriological test optimization

The *mutation operator* is still present in the new model. Since the structure chosen for bacteria is the same as the one chosen for genes, the mutation operator is also the same. On the other hand, 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 converging towards the optimal solution (see discussion in section 4.2).

This approach, as the previous one, needs a *fitness function* to select bacteria that are memorized from one generation to the other. Since the bacterium model is the same as the gene model, the fitness function can be kept. Bacteria are thus selected according to their mutation score.

The two other differences are the emergence of the `Memory` class, and the two associations towards `Mutant` instead of one. The `Bacteriologic` class useses a `Memory` that is the set of the best bacteria that have been saved in previous generations.

On the other hand, a new association towards the `Mutant` class has appeared. In the genetic approach, the algorithm computed the mutation score of individuals on every mutants at each generation. The `Genetic` class thus had only one association towards the `Mutant` class corresponding to the set of all mutants generated from the CUT. Conversely the bacteriological approach aims at avoiding this expensive mutation score computation by saving bacteria from one generation to the other. The mutation score is computed only on mutants that have not been killed in previous generations. This approach thus keeps track of mutants that have been killed and the ones still alive. This explains the presence of two associations from the `Bacteriologic` class towards the `Mutant` class corresponding to the two different sets of mutants.

5.3. New results

Figure 11 shows results of our bacteriological approach for the case study 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 follows:

Size of a system test case. *Let $B=[N_1,...,N_x]$ be a test case for a parser, containing language constructs (nodes of the syntactic tree). The number x of nodes is the size of this bacterium.*

The size of a bacterium is an important parameter. The bigger a bacterium (test case) the longer it takes to run a test case. On the other hand, if bacteria are too small they can not kill mutants, or they kill so few mutants that we need a very large set of bacteria to reach a good mutation score. We conducted several experiments to tune the size of the bacteria. We do not have enough space here to

display the results of these experiments. We looked for a size small enough so that it is not too long to run a test case, but big enough so that a bacteria can contain enough information to kill mutants. We finally chose 15 as a good size for bacteria.
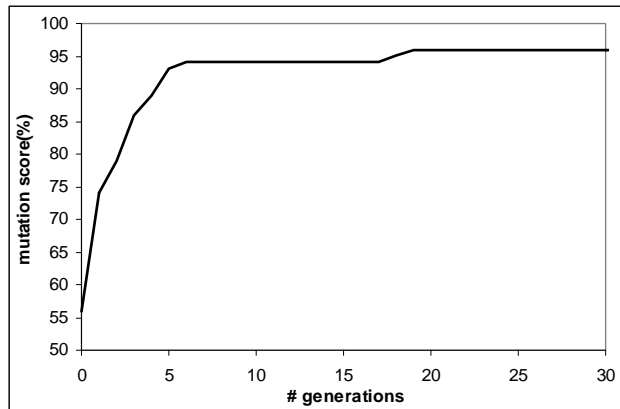

Figure 11 - Results of a bacteriological approach for system test data optimization

This approach converges faster than the previous one. 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. However, since the computation done to go from one generation to the other is not the same in both approach, we give more comparable figures in the column of the table. It gives the number of time a mutant program has been executed. This is a better estimation than the number of generation for the complexity since executing a mutant is as much time-consuming in both approaches.

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

Other interesting results come out of these new experiments. First the memory avoids troughs in the convergence curve and thus speeds up the convergence. A second point is the saving about the tuning effort thanks to the removal of several parameters (size of an individual, selection of individuals for reproduction). This makes the bacteriological approach more reusable for test generation/optimization problems. Removing parameters also makes the model more controllable since there is less random in the algorithm's evolution. The approach is thus more stable than the genetic one.

Two remarks can be made about this model. First, the final set of all the memorized bacteria may not be minimum, for example at the end of the process 9 bacteria

were memorized for the C# parser (this size was 4 in the first model). Second, since the algorithm only saves the best bacterium from one generation to the other, it may miss some information that is present only in weaker bacteria. The minimization can be done in a separate phase after the algorithm has been ran. This step consists in building a boolean matrix which rows are the test cases and the columns the mutants. A 1 in the matrix means that the test case kills the mutant, and a 0 means that it does not. This matrix is called the *coverage matrix* of the mutants by the test cases. This matrix can be minimized to remove redundant information: for example, if the set of mutants killed by a test case is included in the set of another test case, then remove the first test case. This minimizes the result set of test cases. Now, looking at the loss of information due to the memorization of the only best bacteria, a solution could consist in taking a bacterium in the memory set and reinserting it in the new population. For example, one could decide to do this when the mutation score does not improve any more.

As a conclusion about the new experiments, it seems that the adaptations that had been detected as necessary at the end of section 4.2 were actually good heuristics for our problem. This guided us towards a new model, we have called the bacteriological model, based more on an adaptive approach than on the optimization approach. This model seems more stable and reusable for the type of problems we are interested in. It should now be experienced in more details.

6. Related work

While electronic devices have set of measures characterizing their quality (reliability, performance, use-domain, speed scale), no real consensus exists to measure such quality characteristics for software components. Binder details the existing analogy between hardware and OO software testing and suggests an OO testing approach close to the "built-in-test" and "design-for-testability" hardware notions [17]. In this paper, we go even further than Binder suggests, and detail how to create self-testable OO components, with an explicit analogy with the "built-in-self-test" hardware terminology. Moreover, an original measure of the quality of components has been defined based on the quality of their associated tests (itself based on fault injection). For measuring test quality, the presented approach differs from classical mutation analysis [9, 18] by the chosen reduced set of mutation operators.

Besides, the test problem may be seen from a pragmatic point of view, and some simple-to-apply methodology can be found in the literature, which are based on an explicit test philosophy [1]. In this paper, the proposed methodology is based, on a first step, of pragmatic unit test generation and aims at bridging the existing gap between unit and system dynamic tests. In a second step, advanced test optimization techniques, such as genetic algorithms, may help for automatically improving test quality and, consequently, component trustability. To achieve a complete design-for-trust process, the notion of structural test dependencies has been developed for modeling the systematic use of self-testable components for structural system test. In [12], the design-for-testability main methodology is outlined.

Several studies have used genetic algorithms to improve software quality. The Aristotle research group has developed a tool to automatically generate test data based on a genetic algorithm [19]. The tool generates test data that cover a given statement, path, or def-use pair. This work compares genetic algorithms and random process for the test data generation. In [20], genetic algorithms are used in a control-flow coverage-oriented way: test sets are improved to reach such a predefined test adequacy criterion. In [21], genetic algorithms are used to perform some kind of reliability assessment. In this paper, the application of genetic algorithm is coherent with the application of mutation analysis for test qualification. This conceptual continuity, due to the constant analogy of the test selection problem with a "Darwinian" analogy, appears if we consider that the mutation tool allows both the mutation of programs and the mutation of genes (part of a test "individual") via the domain perturbation mutation operator.

Olsson and Runeson tackled the problem of validating system test cases with mutation analysis in [10]. This work focuses on state-based descriptions of software systems. The authors propose mutation operators that can be applied at an abstract level on SDL specifications. These operators model errors that appear because of interactions between elements in the model. These errors can be detected during system testing. The proposed mutation operators can thus validate test cases for system testing.

7. Conclusion

The general assumption for this work is to measure the quality of test cases (the revealing power of the test cases [3]), to build trust in a component passing those test cases. We thus propose a measure of the quality of test cases based on the number of injected faults the test cases can find. Experiments have shown that it is easy to write a set of mean test cases, but that improving this initial set is very difficult and time-consuming. The work presented in this paper tackled the particular issue of automating the improvement of an initial test case that detects around 60% of injected faults so that it can detect more than 90%.

We presented a general framework for faults injection. The qualification of test cases based on faults injection is called mutation analysis and it has been adapted to system

test cases qualification. Based on mutation analysis to estimate the quality of a tests cases set, we experimented two different models for test optimization. First we computed genetic algorithms to improve an initial set. We modeled the test optimization problem so that it could fit genetic algorithms and ran experiments on a C# case study. The results of these experiments were deceiving because the test cases quality increased very slowly and did not reach very high values. A new model, called bacteriological model, simulates the bacteriological adaptation phenomenon. Conversely to genetic algorithms, this approach optimizes test cases and not a test cases set, and this new model memorizes efficient test cases from one generation to the other. We ran new experiments on the same case study to investigate the improvement of test cases quality.

References

[1] K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests". Java Report. Vol, 1998.

[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] 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.

[4] 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.

[5] 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.

[6] I. Moore. "Jester - a JUnit test tester". in proceedings of *XP'2001*, Villasimius, Sardinia2001.

[7] MSDN. ".NET homepage"

http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000519

[8] MSDN. "C# Introduction and Overview" http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpintro.asp

[9] A.J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing". Software Practice and Experience. Vol.26(2), 1996.

[10] 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.

[11] B. Meyer, "Object-oriented software construction", Prentice Hall, 1992.

[12] Y. Le Traon, D. Deveaux, and J.-M. Jézéquel. "Self-testable components: from pragmatic tests to a design-for-testability methodology". in proceedings of *TOOLS' Europe*, Nancy, France, June 1999.

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

[14] J.H. Holland, "Adaptation in Natural and Artificial Systems", University of Michigan Press, 1974.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software". Professional Computing, Addison-Wesley, 1995.

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

[17] R.V. Binder, "Testing Object-Oriented Systems: Models, Patterns and Tools", Addison-Wesley, 1999.

[18] R. DeMillo and A.J. Offutt, "Constraint-Based Automatic Test Data Generation". IEEE Transactions on Software Enginnering. Vol.17(9), p. 900-910, 1991.

[19] R. Pargas, M.J. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms". Journal of Software Testing, Verifications, and Reliability. Vol.9, p. 263-283, 1999.

[20] B.F. Jones, H.-H. Sthamer, and D.E. Eyres, "Automatic Structural Testing Using Genetic Algorithms". Software Engineering Journal. Vol.11(5), p. 299-306, 1996.

[21] S.A. Wadekar and S.S. Gokhale. "Exploring Cost and Reliability Tradeoffs in Architectural Alternatives Using a Genetic Algorithm". in proceedings of *ISSRE (International Symposium on Software Reliability Engineering)*, Boca Raton, Florida, November 1999.

Appendix A : example for C#

Figure 12 gives an example of bacterium (or gene) written in C#. This is an example of C# source file that can be passed as an input to the C# parser. This file contains 20 nodes from the syntactic tree (C# constructs).

The figure also illustrates the mutation operator. The bold `foreach` node in the left source file has been chosen for mutation. A new source file has been created (right hand-side) in which the node has been replaced by a `while` node (bold in the right source file).

```
using System;                                          using System;
namespace Id_1 {                                       namespace Id_1 {
  using System;                                           using System;
  protected class Id_2 {                                  protected class Id_2 {

    [AnAttribute1; AnAttribute2]                           [AnAttribute1; AnAttribute2]
    public string aField;                                  public string aField;

    public ~Id_2() {} //~Id_2                              public ~Id_2() {} //~Id_2

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

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

    [AnAttribute]                                          [AnAttribute]
    static Type aProperty {                                static Type aProperty {
      get {}                                                 get {}
      set {                                                  set {
        aVariable = aValue + 3;                                aVariable = aValue + 3;
        for (int i=0 ; !Id_6||Id_8!=Id_3 ; i++)                for (int i=0 ; !Id_6||Id_8!=Id_3 ; i++)
        {                                                      {
          foreach (nodes n in the_tree)                          while(cond1){
            {anObject.aMethod (param3, param4 );}                    aVariable1++;}
        }                                                      }
      }                                                      }
    }                                                      }

    public returnType1 aMethod2 (Type3 param5) {} //aMethod2   public returnType1 aMethod2 (Type3 param5) {} //aMethod2
  } //Id_2                                                } //Id_2
}                                                      }
```

Figure 12 - example of a bacterium (or a gene) for C# parser