SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2005; **15**:73–96
Published online 4 January 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.313

From genetic to bacteriological algorithms for mutation-based testing‡

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

IRISA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France

SUMMARY

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: faults are injected into the software component (making mutants of it) to check the proportion of mutants detected ('killed') by the test cases. But while the generation of a set of basic test cases is easy, improving its quality may require prohibitive effort. This paper focuses on the issue of automating the test optimization. The application of genetic algorithms would appear to be an interesting way of tackling it. The optimization problem is modelled 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, which is the test cases set provided by the programmer. To overcome disappointing experimentation results, on .Net components and unit Eiffel classes, a slight variation on this idea is studied, no longer at the 'animal' level (lions killing zebras, say) 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 operator. The purpose of this paper is to explain how the genetic algorithms have been adapted to fit with the issue of test optimization. The resulting algorithm differs so much from genetic algorithms that it has been given another name: bacteriological algorithm. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: automatic test generation; evolutionist algorithms; object-oriented testing; mutation analysis

*Correspondence to: Benoit Baudry, IRISA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France.
†E-mail: bbaudry@irisa.fr

‡Based on 'Genes and bacteria for automatic test cases optimization in the .NET environment' by Benoit Baudry, Frank Fleurey, Jean-Marc Jézéquel and Yves Le Traon which appeared in *Proceedings of the International Symposium on Software Reliability Engineering*, Annapolis, MD, November 2002, pp. 195–206 [1]. © 2002 IEEE. This revised and expanded version appears here with the permission of the IEEE.

Received 24 November 2003
Accepted 18 June 2004

1. INTRODUCTION

Some specialists have claimed, 'Programmers love writing tests' [2]. 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 in a given software component is then linked to the quality of its test cases. Conversely, one way to qualify the test cases consists of deliberately introducing faults in the software under test. The intuition for this technique, called *mutation analysis* [3], 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 into the original implementation. By measuring the quality of test cases (the revealing power of the test cases [4]), trust is built in a component passing those test cases. Mutation analysis has been successfully applied in the quality assessment of unit test cases for object-oriented (OO) classes [5–7], and gives the programmer interesting feedback on the 'revealing power' of his/her test cases. It also offers an estimate of how many new test cases are needed to test a given software component better.

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

The issue of automatically improving test cases is a nonlinear optimization problem, and the application of genetic algorithms (GAs) appears to be an interesting way to tackle 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 in the optimization. In this paper, the optimization problem is modelled 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, which is the set of test cases provided by the tester. The adaptation of GAs to this context is presented here, as well as the analysis of the results obtained with two case studies: one at the unit test cases level (for Eiffel classes) and the other at the system level testing (the testing of a C# parser in the .Net framework [8,9]). While it was quite disappointing that these experimental results were not as good as expected, biologist colleagues suggested trying a slight variation on this idea, no longer at the 'animal' level (lions killing zebras, say) 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 and the notion of individuals (genotype). The main contribution of this paper concerns the way the GAs have been adapted to propose a novel algorithm: the *bacteriological algorithm* (BA). The bacteriological model and its behaviour are described and validated using the previous case studies.

The rest of this paper is organized as follows. Section 2 opens with a brief summary about mutation analysis, and then introduces how it is adapted to test generation and optimization. Mutation analysis has never been applied to system testing, because of prohibitive execution times. A derived contribution of this paper concerns the flexibility of the mutation approach either to a single class or to a whole system. Section 3 presents a model for test optimization that builds on GAs. Section 4 presents two case studies that have 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, and new results for both case studies. In Section 6 some related work is discussed and Section 7 gives several conclusions about this work.

2. MUTATION TESTING FOR THE OO DOMAIN

Mutation testing is a testing technique which was first designed to create effective test data, with an important fault revealing power [4,10]. It was originally proposed in 1978 [3], and consists of 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 modelled 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.

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 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:

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

It is to be noted that a mutant is considered *equivalent* to the original program if there are no input data at all on which the mutant and the original program produce a different output. 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.

The remainder of this section is organized as follows. The general test selection process based on mutation analysis and the chosen mutation operators are presented. Then, a generic framework is described for unit and system test cases optimization based on a common mutation core.

2.1. Test selection process

The whole process for generating test cases with fault injection is presented in Figure 1(a). It includes the generation of mutants from the Component Under Test (CUT) and the application of test cases against each mutant. Two types of oracle can be used to kill mutants:

- the difference between the result of the initial implementation and the mutant result;
- contracts, as an embedded oracle function, derived from the specification in a design-by-contract approach [11].

In Figure 1, 'diagnosis' designates a non-automated task which aims at determining why a mutant is alive after the execution of test cases: it may be due to the test cases, to an incomplete specification (and particularly if contracts are used as oracle functions) or to the fact that the mutant is equivalent. In the first case, it means that the set of test cases is weak, and that new test cases must be added to increase the mutation score. In the second case, it means that the embedded oracle functions are weak, and that the assertions (pre/postconditions, invariants) have to be reinforced. It has to be noted that when the set of test cases is selected, the mutation score is fixed as well as the test quality of the component. Moreover, except for the diagnosis, the process is completely automated.

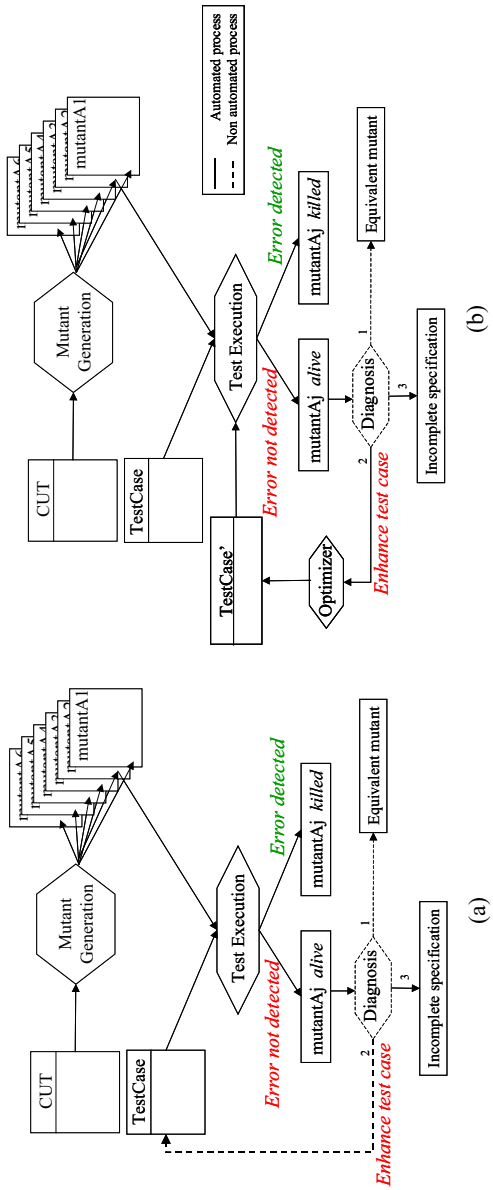

Figure 1. The mutation process.

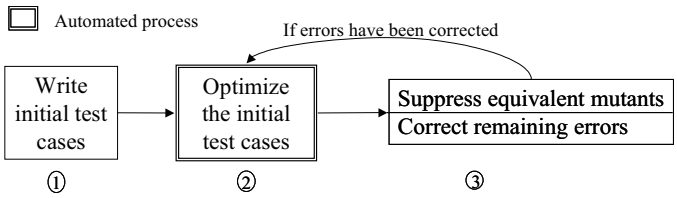

Figure 2. Incremental process for software testing.

The assumption when applying mutation is that the original program passes its initial set of test cases (detected bugs have been corrected). Then test case optimization aims at enhancing this set so that it has a high mutation score. At the end of the process, the new test cases must be applied on the original program to check if they detect a fault. This is the reason why as few test cases as possible must be added since they must still be associated with an oracle and applied to the original program: there is a supplementary cost due to the determination of the oracle for the original program. This paper focuses on the test optimization process to obtain automatically 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). This corresponds to the automation of the test case enhancement phase after the diagnosis in the mutation process. In Figure 1(b) an 'optimizer' operation has appeared that optimizes the initial test case to improve its mutation score. Different strategies have been investigated to automate the 'optimizer' operation: genetic algorithms (see Section 3) or an adaptation of these algorithms called bacteriological algorithms (see Section 5).

In [6], a testing-for-trust methodology was proposed, 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, the focus is on test generation/optimization and the corresponding stages are extracted from the global methodology. Based on the process of Figure 1(b), Figure 2 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.

2.2. Mutation operators

The set of mutation operators should:

- be general enough to be applied to various OO languages (Java, C++, Eiffel, etc.);
- imply a limited computational expense;
- ensure at least control-flow coverage of methods.

Table I. Mutation operators set for OO programs.

Type	Description
EHF	Exception Handling Fault
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCP	Variable and Constant Perturbation
MCR	Methods Call Replacement
RFI	Referencing Fault Insertion

The actual choice of mutation operators includes selective relational and arithmetic operator replacement, variable perturbation, but also referencing faults (aliasing errors) for declared objects. The choice of mutation operators is given in Table I.

2.2.1. Functionality of each of the mutation operators

EHF: causes an exception when executed. This semantically large mutation operator forces code coverage.
AOR: replaces occurrences of '+' by '−' and other similar replacements as follows:

Arithmetic operator	Replaced by
+	−, *
−	+, / (or div)
*	/ (or div), +
/	*, −
div	−, mod
mod	−, div

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.
ROR: each occurrence of one of the relational operators (<, >, <=, >=, =, /=) is replaced by each one of the other operators.
NOR: replaces each statement by the *Null* statement.
VCP: constant and variable values are slightly modified to emulate domain perturbation testing. Each constant or variable of arithmetic type is both incremented by one and decremented by one. Each *Boolean* is replaced by its complement.
MCR: method calls are replaced by a call to another method with the same signature.
RFI: forces the reference to an object to be stuck at null after its creation. Remove a clone or copy instruction. Insert a clone instruction for each reference assignment.

The mutation operators AOR, LOR, ROR and NOR are traditional mutation operators [10,12,13], but the other operators have been introduced in this paper for the OO domain. The data perturbation operator VCP allows one to disturb the state of data and to obtain a sensitivity analysis of a program in a manner similar to the approach of Voas and Miller [4]. Operator RFI introduces object aliasing and object reference faults, specific to OO programming:

- reference to each object is stuck-at null;
- object duplication instructions (**clone/copy**) are removed;
- each assignment of an object is preceded by the duplication of this object.

The faults due to the RFI operator are more difficult to detect than those due to other operators, since it forces the test cases to detect that some data structures are not owned by the specified objects.

2.3. Mutation for unit and system testing

In this section, the issue of mutation for a system composed of a set of unit classes is tackled and a pragmatic solution is proposed. Mutation analysis has never been applied to global system testing. In the case of unit testing, the faults are injected into a single class under test. With system testing, faults are injected into all the components of the system, a mutant system being a system in which a single fault has been injected as for a unit class mutant. Since the purposes of unit and system testing are different, the mutation approach must be adapted and considered in a different way. At the system level, the fundamental assumption for mutation, that is the 'Competent Programmer Hypothesis', has to be reformulated as the 'Competent Designer Hypothesis': mutation operators specific to design faults should be proposed. Since the complexity of a system is higher than a single class, two main specific issues for using mutation at the system level appear.

1. *Combinatorial explosion of the number of mutants*: while it is reasonable to inject many different types of faults in a class, it is not realistic to inject all the possible faults into the system. The execution/compilation time for applying all the test cases on a mutant system is much greater than on a unit class.
2. *Determination of equivalent system mutants*: although mutant equivalence is often decidable on a class, it may not be possible for a tester to decide system equivalence.

Considering these issues, the solution proposed in this paper is pragmatic and aims at reusing the unit level results and operators for system testing. A methodology based on mutation testing consists of testing unit classes using mutation before the mutation is applied at the system level. When performing system testing, classes are expected to have been successfully tested at the unit level (with respect to the whole set of mutation operators). System testing then focuses on the relationships between the classes in the system, i.e. the structural design (class diagram). In consequence, one may choose a subset of existing operators to perform system testing. Second, equivalent mutants are avoided so that the second point above is no longer an issue. The LOR and NOR operators should generate only non-equivalent mutant systems, since any terms of a logical expression and any statement should have an impact on the system result (or it may correspond to dead code that must be suppressed). By choosing a subset of the unit mutation operators, the combinatorial explosion is avoided for systems of medium size. This is the pragmatic solution that has been applied in this work.

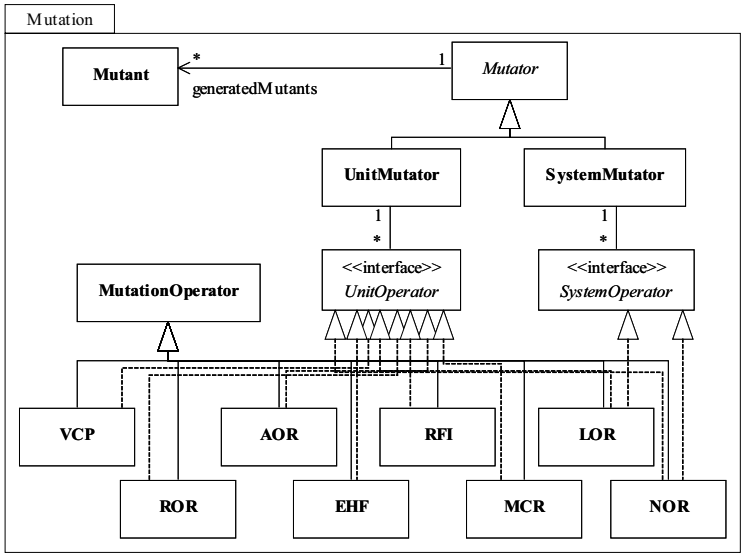

Figure 3. A generic model for the mutation tool.

To make the concepts clearer and show how unit mutation operators are reused at the system level, Figure 3 presents the generic UML model for the two variants (unit/system) of the mutation tool. For unit testing, the operators that implement the `UnitTesting` interface (all) are applied; in the same way, for system testing, operators that implement the `SystemTesting` interface (LOR and NOR) are available.

The objective of this paper is not to solve the problem of mutation for a system but to show that GAs can be adapted to optimize a set of test cases both for unit classes and for a set of interconnected classes (called 'system' in the context of this paper).

3. TEST CASES GENERATION: GENETIC ALGORITHMS FOR TEST GENERATION

Writing a first set of test cases is easy, and most developers do such basic testing. Experiments have shown that such test cases easily reach 60% of test quality (see [14]). Improving test quality implies a particular and specific supplementary testing effort. In this section the use of GAs is investigated as a pragmatic way of automatically improving the basic set of test cases 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 there is a population of mutant programs to be killed and a test cases pool. Those test cases (or 'gene pool') are randomly combined to build an initial population of test

cases seen as predators of the mutant population. A GA is applied to improve the ability of this initial population to kill mutant programs.

3.1. Genetic algorithms

GAs [15] were first developed by John Holland [16], whose goal was to explain natural systems rigorously and then design artificial systems based on natural mechanisms. So, GAs 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, etc.) reproduce and, thanks to crossover and mutation, the next generation will fit better. This is just how a GA works: it uses an objective criterion to select the fittest individuals in one population, it copies them and creates new individuals with pieces of the old ones.

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

To apply GAs to a particular problem, it has to be decomposed into atomic units that correspond to genes. Then individuals can be built, 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 $F(x)$, the value of which is the quality of the individual regarding the problem to solve. This corresponds to the function that has to be maximized.

Moreover, a GA uses three operators: reproduction, crossover, mutation.

- *Reproduction* 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. The wheel is spun as many times as the size of the population, and so a new population is available, which is going to participate in crossover. This new population is made of individuals from the old population, 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, one chooses, at random, an integer value k between 0 and the size n of an individual less one. Second, one creates two new individuals A' and B' from a pair (A, B); A' consists of the first k genes of A and the last $n - k$ genes of B, and B' consists of the first k genes of B and the last $n - k$ genes of A.
- The *mutation* operator modifies the values of one or several genes (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 GA is computed following the process described in Figure 4.

3.2. Genetic algorithms for test optimization

This section presents an adaptation of a GA to optimize a set of initial tests automatically, based on mutation score as a quality criterion. First, a generic model that can be applied to optimize any type of test data is presented. Then specializations for unit and system testing are discussed.

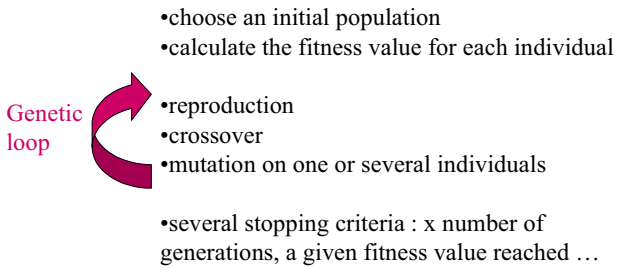

Figure 4. The global process of a GA.

$$\text{ind}_1 = \{G_{1\,1}, \ldots, G_{1\,i}, G_{1\,i+1}, \ldots, G_{1\,m}\} \qquad \text{ind}_2 = \{G_{2\,1}, \ldots, G_{2\,i}, G_{2\,i+1}, \ldots, G_{2\,m}\}$$

$$\text{ind}_3 = \{G_{1\,1}, \ldots, G_{1\,i}, G_{2\,i+1}, \ldots, G_{2\,m}\} \qquad \text{ind}_4 = \{G_{2\,1}, \ldots, G_{2\,i}, G_{1\,i+1}, \ldots, G_{1\,m}\}$$

Figure 5. The crossover operator.

The points discussed here are specific adaptations of the genetic model to test optimization that strongly depend on the type of test data that are improved.

3.2.1. Modelling reproduction and crossover

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 GA. In the case of test optimization, a *gene* corresponds to a *test case*. However, Sections 3.2.2 and 3.2.3 will show that a test case is not represented in the same way depending on the testing level (unit or system). The *mutation score* associated with a test case corresponds to its *fitness* value.

The reproduction and crossover operations can be expressed at a generic level. Those two operators are independent from a specific gene model, and are thus independent from a particular component under test. Conversely, the mutation operator is very dependent on a particular gene model and will thus be defined separately for unit and system testing in the following subsections.

- *Reproduction*: the slot for each individual in the roulette wheel is proportional to its mutation score.
- *Crossover*: let m be the size of an individual, and select an integer i at random between 1 and $m - 1$. Then from two individuals ind_1 and ind_2, two new individuals ind_3 and ind_4 are created; one made of the first i genes of ind_1 and the last $m - i$ genes of ind_2, and the other made of the first i genes of ind_2 and the last $m - i$ genes of ind_1. This operator is illustrated in Figure 5.

The next subsections detail unit or system-dependent aspects of the genetic modelling for test optimization: the gene model and the associated mutation operator. The gene model also has to be correct according to the other operators; in particular, it has to permit the usage of the crossover operator. This means that wherever genes are located inside an individual, this individual must be a correct set of inputs for the CUT.

3.2.2. Specialization for unit testing

A unit test case is a method which creates one or several instances of the class under test, and calls methods on these objects. This way of writing unit test cases has been standardized with the emergence of the XUnit test framework family.

To apply GAs to unit test cases optimization, a gene is thus modelled as a method. Two parts are clearly identified in this method as detailed in the following definition.

Gene modelling for unit testing. A gene is a test case for a unit. It is modelled as a method composed of two parts:

1. creation and initialization of objects that are to be tested;
2. method calls on these objects.

Let $m_1, \ldots, m_n$ be n method calls and $p_1, \ldots, p_n$ instances of the parameters for the method calls. A gene is denoted $G = [I, S]$ where I is a sequence of statements that initialize the object under test and $S = (m_1(p_1), \ldots, m_n(p_n))$.

Based on this model, the mutation operator consists of changing the value of the parameters for one method call in the set S of one gene.

Mutation operator for unit testing. The mutation operator changes the parameter values of one method call in one gene, as illustrated below.

$$G = [I, S] \Rightarrow G = [I, S_{\text{mut}}]$$

$$S = (m_1(p_1), \ldots, m_i(p_i), \ldots, m_n(p_n)) \Rightarrow S_{\text{mut}} = (m_1(p_1), \ldots, m_i(p_{\text{imut}}), \ldots, m_n(p_n))$$

This mutation operator is important for control-flow coverage.

Concrete examples of a source file and the mutation operator are given in Appendix A.

3.2.3. Specialization for system testing

The gene model and mutation operator described in this section strongly depend on the case study for system testing: a parser. For this particular system, the input data consist of a source file that is parsed to build a syntax tree. The gene model is given in the following definition.

Gene modelling for system testing. 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 syntax tree). If there are x nodes in the file a gene can be represented as follows:

$$G = [N_1, \ldots, N_x]$$

Based on this gene modelling, the mutation operator consists of replacing a syntax node in a source file (an individual) by another valid node with respect to the grammar of the language.

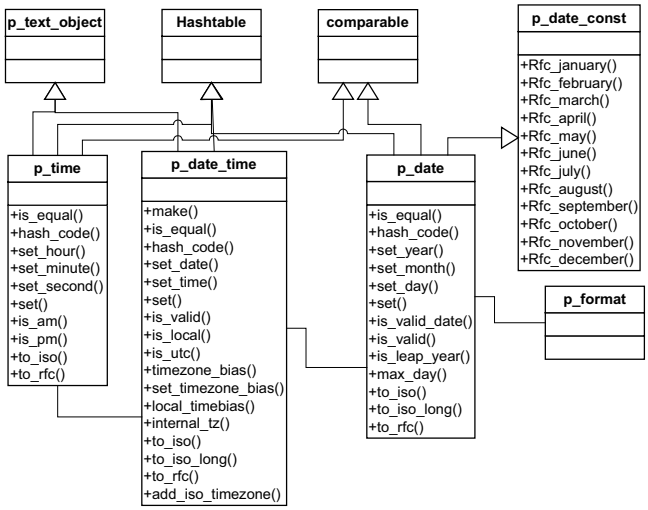

Figure 6. Classes of package 'date-time'.

Mutation operator for system testing. The mutation operator, chooses a gene at random in an individual and replaces a node in that gene by another one:

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

Concrete examples of a source file and the mutation operator are given in Appendix B.

4. CASE STUDIES WITH GENETIC ALGORITHMS

This section describes two case studies that have been conducted to study the automation of test cases optimization using a GA. The two case studies concentrate on different levels of testing. The first one concerns unit testing and is based on an Eiffel library. The second one applies a GA to optimize tests for a small system written in C# in the .NET framework. The two case studies have been chosen to represent classical categories of software. The classes studied are typical classes, since methods are small and manipulate few data. The .Net component is typical of any software that transforms input data in a given format into a new format. For instance, the same type of model for optimization can be used for testing software that uses XML as an exchange format.

4.1. Unit test data optimization: an Eiffel example

The case study for unit testing is based on the Pylon library. It is a small, portable, freely available Eiffel library for data structures and other basic features. The experiments focus on three classes

Table II. Mutation scores for initial test sets.

	p_date	p_date_time	p_time
No. of generated mutants	673	199	275
Mutation score (%)	53	58	58

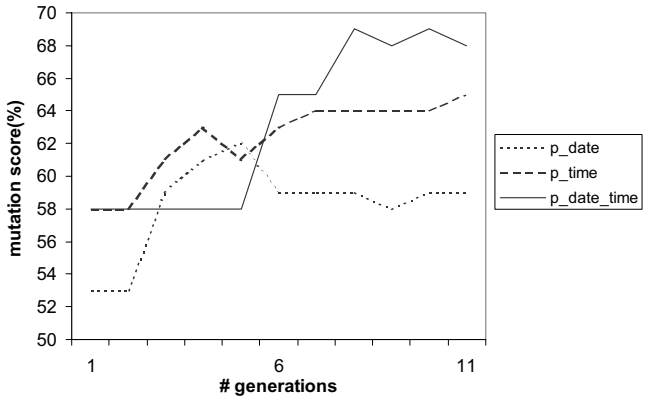

Figure 7. GA application for unit test data optimization.

from the package that deals with time and date management. The main class of this package is called `p_date_time.e`. The way in which the various classes used in this package interact is presented in Figure 6.

An initial test set has been written for each of the three classes `p_date`, `p_date_time` and `p_time`. Then, the mutation scores of these three test sets were computed. The results are summarized in Table II. The differences for the number of mutants generated are due to the differences in complexity of the methods in the classes. The number of mutants is proportional to the number of lines of code, the number of control points, as well as the number of predicates in the logic expressions at the control points. For example, there are far fewer mutants for `p_date_time`, because most of the methods of this class delegate their computation to methods of classes `p_time` and `p_date`.

These initial test sets correspond to the test seed that can be used for automatic improvement through GAs. The algorithm was run to improve the three test sets. In each case, the initial test sets included three test cases that concentrated on different behaviours of the associated class. Those test cases were used as genes to initialize the population to be improved. The population consisted of 15 individuals, each one containing 5 genes. The mutation rate was 10%.

Figure 7 presents the curves of the mutation score as a function of the number of generated predators (one point represents a generation step).

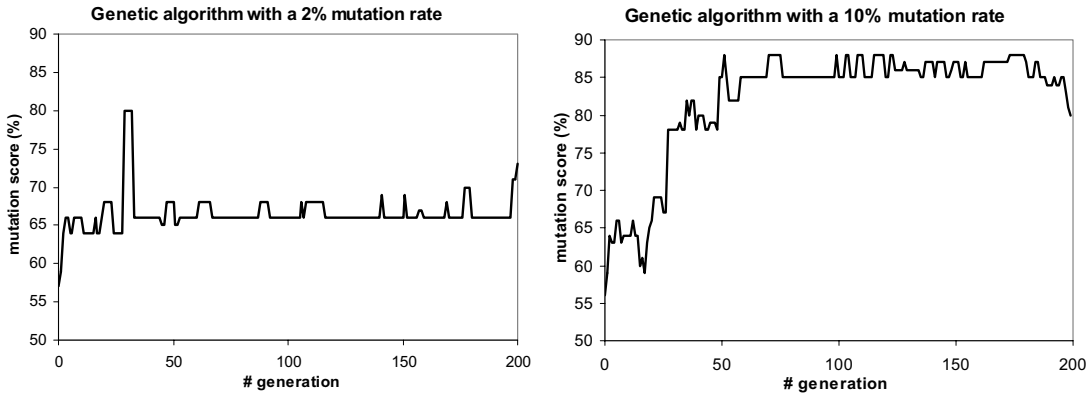

Figure 8. GA application for test optimization for a C# parser.

4.2. System test data optimization: testing a .NET component

For system testing, the test data optimization technique has been applied on a .Net component that parses C# source files. There are 32 classes in this system that is implemented in C#. This parser takes a set of C# source files as input and builds the corresponding syntax tree.

To experiment with GAs on this system, 500 mutant systems were generated, using only the NOR operator. Nevertheless, the results obtained are still interesting since the test cases generated against such mutants cover all statements in the system. The initial population for the GA application consisted of 12 individuals of size 4, and its initial mutation score was 56%. The results are given in Figure 8.

4.3. Results and comments

This section summarizes several conclusions about the application of a GA to improve the quality of test cases (Figures 7 and 8). The results are discussed in terms of fault detection capacity, growth of the mutation score, model tuning, and execution time.

In terms of fault detection. Several errors were found and corrected. Alive mutants were also studied. Some mutants were obviously not equivalent, but still alive, and they actually corresponded to errors that had been injected in dead code.

In terms of mutation score. Even if the GA automatically improved the mutation score of the set of initial test cases, this improvement was not satisfactory: either because of the small improvement in the case of unit test cases, or, in general, because of the slow convergence and the unusual proportions of crossover and mutation operators. To go from one generation to another, GAs 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.

Critical information loss appears when passing from one generation to the next. In that case, the best individual of the new population may be worse than the best one of the previous generation. This phenomenon implies a slow convergence. *Memorizing the individuals before reproduction would solve this problem.*

In terms of model tuning. The first parameter that is analysed here is the size of an individual. GAs look for an optimal individual, not an optimal population. Thus, an individual has to be equivalent to a set of tests in the particular case of test optimization. It is very difficult to predict how many test cases will be necessary to kill every mutant for a particular program. So, the size has to be set to a high value at the beginning, and then tuned, so that the final individual has a good mutation score but is not too big. Big sets are not interesting because then running all the test cases is much too time-consuming. The tuning has to be done for every particular CUT. Even if this tuning is mandatory when applying GAs to a particular problem, it seems particularly constraining in the case treated here since the objective is to improve test cases and not an individual. *The goal is to have good test cases with no strong constraint on the number of tests. Thus, a better adapted model would not constrain the size of the set when improving the test cases.*

The second parameter is the mutation rate. The mutation rate had to be excessively increased compared with usual application of genetic algorithms. Figure 8 shows results with two different mutation rates: 2% and 10%. The lowest rate gives no useful result, since the mutation score reaches at most 80%, whereas the 10% rate makes the mutation score increase to almost 90%. Actually, it appears that the mutation operator is the one that creates information. In both cases (unit or system) this operator changes the test data. So after mutation, the test case might cover other parts of the CUT. For test optimization, this represents an information saving. The mutation rate was thus set to 10% for the experiments.

The last parameter is the crossover operator. The limitation of this operator is not so much the tuning, but the lack of efficiency in the case of OO testing. Indeed, the way genes are modelled 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 has no importance. This makes the crossover operator useless, since its only function is to create information by reordering genes inside an individual.

In terms of execution time. A total of 480 000 mutant systems had to be executed to reach a mutation score of 85%. With a 1.33 GHz bi-processor with two mutant executions in parallel, a mutant system needed 0.4 s to be executed: the global execution time for the GA on a single computer was 26 h. This is not efficient, even if mutant executions could be launched in parallel on several machines.

As a conclusion about these experiments, crossover appears as being not perfectly adapted to the test cases optimization problem. A better adapted model should provide memory and remove the notion of individuals to concentrate on the genes (test cases). This would avoid tuning when applying the model on different CUTs. Nevertheless, some things can be retained from this experience: (1) the gene modelling which is clearly defined and corresponds exactly to what has to be optimized; (2) the mutation operator that seems to be a good way of creating new information in the context of OO test generation; (3) the mutation score as the fitness function that guides the algorithm towards a good solution. The next section proposes a new model and process, adapted from the GAs and based on these conclusions. It is called the bacteriological approach, and is based on the bacteriological adaptation phenomenon.

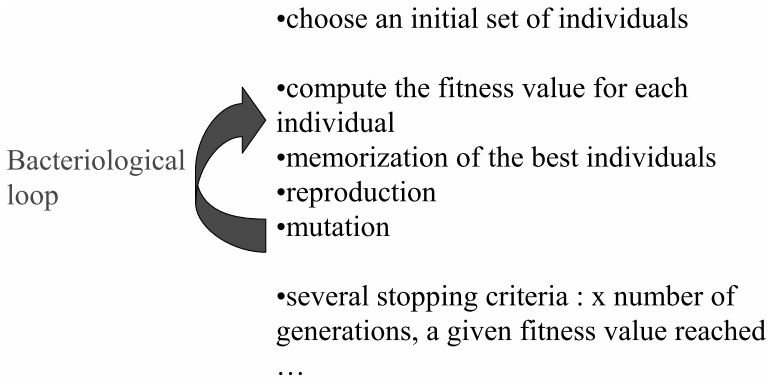

Figure 9. The bacteriological process.

5. AN ADAPTIVE APPROACH: BACTERIOLOGICAL ALGORITHMS

The experiments described in Section 4 have shown some drawbacks of GAs for test case optimization. This section presents an adaptation of the genetic approach for OO test generation. The adaptation consists of keeping track of the best individuals from one generation to the next. 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 decreases at each step of the genetic loop with the size of the alive mutants set.

Even if the adaptation of the genetic model appears to be based on very small modifications to the process, it actually completely changes the idea of the GA, 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 next, since the goal of the search (killing every alive mutant) changes at each generation. Moreover, the new model does not generate the optimal individual, but a set of individuals (the ones that have been memorized during the whole process). The new approach is thus quite far from the genetic model. Keeping the analogy with biological processes, this new model is close to 'bacteriologic adaptation' [17].

5.1. The bacteriological model

5.1.1. The global process

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

As with the genetic model, a *fitness function* is necessary to choose bacteria for reproduction. With this function a global iterative process to adapt an initial population is given in 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 generations or when the memorized population has reached an optimum fitness value.

5.1.2. The model for test optimization

A bacterium is modelled as a test case.

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 in Sections 3.2.2 and 3.2.3, the mutation operators are also the same. On the other hand, since this approach only manipulates bacteria which correspond to genes in the previous approach, the reproduction and crossover operators have disappeared. The removal of the crossover operation is one major difference with the genetic model.

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

The other difference is the emergence of the memorization. The bacteriological approach manipulates a memory that is the set of the best bacteria that have been saved in previous generations. In the genetic approach, the algorithm computed the mutation score of individuals on every mutant at each generation. Conversely, the bacteriological approach aims at avoiding this expensive mutation score computation by saving bacteria from one generation to another. 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.

5.2. New results

5.2.1. Experiments

Figures 10 and 11 show results of the 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 in both cases (between three and ten bacteria), the experiments were conducted by saving only the best bacterium for a given generation.

The size of a bacterium is defined in a different way depending on the type of test case.

Size of a unit test case. Let $B = [I, S]$ be a unit test case, where I is an initialization sequence and S is a set of method calls on instances of the CUT. The size of this bacterium is the size of S.

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

Extensive experimental work concerning the tuning of this parameter is presented elsewhere [18].

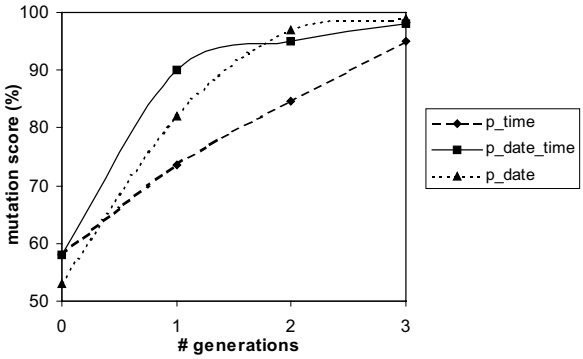

Figure 10. Results of a bacteriological approach for unit test data optimization.

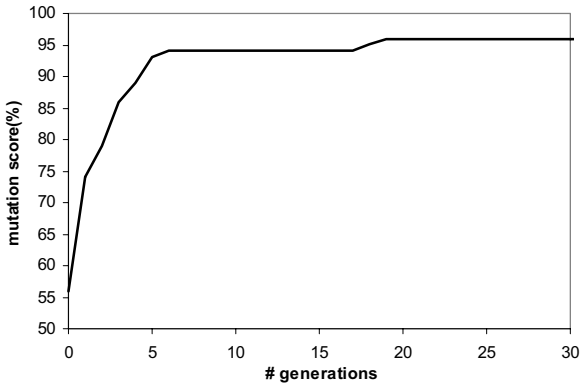

Figure 11. Results of a bacteriological approach for system test data optimization.

5.2.2. Discussion

This approach converges faster than the previous one. Table III summarizes the results of both approaches for the C# parser. This table gives the number of generations needed to reach the score given in the third column. The bacteriological algorithm converges much faster than the genetic one: 30 generations instead of 200. However, since the computation to go from one generation to the other is not the same in both approaches, more comparable figures are given in the last column of the table. It gives the number of times a mutant system has been executed. This is a better estimate than the number of generations for the complexity since executing a mutant is equally time consuming in both approaches. With a 1.33 GHz bi-processor with two mutant executions in parallel, a mutant system

Table III. Comparison between genetic and bacteriological algorithms for the C# parser.

Algorithm	No. generations	Mutation score (%)	No. mutants executed
Genetic	200	85	480 000
Bacteriologic	30	96	46 375

needed 0.4 s to be executed: the global execution time for the GA on a single computer was 26 h and for the BA it was 2 h 34 min. Since the process is completely automated, it seems reasonable to have some hours to wait for test case optimization (compared with the human effort that would be necessary to reach the same mutation score).

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 on 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 randomness 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, nine bacteria were memorized for the C# parser. Second, since the algorithm only saves the best bacterium from one generation to another, 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 run. This step consists of building a Boolean matrix whose rows are the test cases and whose columns are 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 only the best bacteria, a solution could consist of 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.

The new results show that the adaptations that had been detected as necessary at the end of Section 4.3 were actually good heuristics. These observations led towards a new model, called the bacteriological model, based more on an adaptive approach than on the optimization approach. This model seems more stable and reusable.

More work is currently being undertaken with regard to this algorithm. In particular, the impact of the different parameters on the evolution of the fitness value and on the convergence speed is being carefully studied. Another parameter, called memorization threshold, has also been introduced, which defines a fitness threshold value above which a bacterium can be memorized. Indeed, as it is now, if a bacterium is good enough to improve the solution set, it is memorized. The idea with the memorization threshold is to wait until the bacterium's fitness reaches the threshold before memorizing it. In that way, better bacteria are memorized, and the final solution set contains fewer bacteria for the same fitness.

6. RELATED WORK

When the studies presented in this paper were performed, no published work existed on mutation operators specific to OO programs and generic enough to be applied to several languages. Since then, Ma *et al.* [19] have proposed specific operators to validate inter-class test cases. This work introduces original operators and also synthesizes operators proposed in other work by Chevalley [20]. Alexander *et al.* [21] also propose OO-specific mutation operators dedicated to the Java language.

Though it was not the primary goal of this work, one issue had to be solved during this study: how can mutation testing be used on a larger scale than unit testing? The chosen solution is a very pragmatic one, and is not a general solution to the subject. Several other works have tackled the issue of mutation for purposes other than unit testing, for example, interface mutation. Interface mutation has been studied to validate the efficiency of test cases for integration of component-based systems. Most component models present the component as a black-box with a public (white-box) interface. Ghosh and Mathur [22] proposed a set of mutation operators that can be applied to methods proposed in a component's interface. The operators of [22] are dedicated to CORBA components, and the paper proposes a comparison, in terms of fault revealing power, between interface mutation and control-flow criteria. Interface mutation has also been used by Yoon and Choi [23] to validate integration testing for EJB components.

Several works have studied GAs for automatically generating software test data. All these works actually map the problem of test data generation to the problem of function minimization and study GAs to solve this minimization problem. The function that has to be minimized depends on the testing criterion that is chosen.

In [24], Michael *et al.* present their testing tool GADGET (Genetic Algorithm Data GEneration Tool). They detail the modelling of the problem to fit the genetic approach and then give experimental results for several programs. For the experiments, the authors use two different GAs and compare the results obtained with random generation and an algorithm based on gradient descent to solve the function minimization problem. In each case, the testing criterion is based on branch coverage. Random generation is efficient only for small programs. For bigger programs, other techniques give better results, and are worthwhile even if more effort is needed to model and tune the models. In either case, the classical GA gives the best results.

In [25], Pargas *et al.* use a fitness function based on the coverage of the control-dependence graph (CDG). They also built a prototype tool, called TGen, to experiment with their approach. This tool generates test data for each test objective defined on the CDG. The GA gives better results for all six programs that were tested.

In [26], Jones *et al.* presented a tool that generates test data which covers a given statement, path, or def-use pair. This work compares GAs and a random process for test data generation.

The major difference between all these works and the studies presented in this paper is that they are interested in generating scalar data, whereas method calls on an object are generated in this paper. Actually, GAs are much more appropriate for generating scalar data: in that case, individuals can be modelled as a byte string, and classical genetic operators are much more efficient under these circumstances. Whereas, a set of test cases is needed in the context of OO testing, each test case being a complex entity (a method or a set of commands) and not just a byte. As explained previously by the current authors [1], and in more detail in this paper, the genetic model is less efficient for OO testing, and it has to be adapted.

7. CONCLUSION

The work presented in this paper tackled the particular issue of automating the improvement of the mutation score of an initial set of test cases. Two different models for test optimization have been studied. First, the problem has been modelled by applying a GA to improve an initial set. Two different case studies have been performed with this first model. The results of these experiments were deceiving because the quality of the test cases increased very slowly and did not reach very high values. The second and novel model, called the bacteriological model, simulates the bacteriological adaptation phenomenon. In contrast to GAs, this approach generates test cases instead of a set of test cases, and memorizes efficient test cases from one generation to the next. New experiments were performed on the same case studies to study the improvement in quality of test cases. Results have shown that this bacteriological model is promising both for the mutation score and computational expense (the average execution time is divided by 10).

APPENDIX A. EXAMPLE FOR EIFFEL

Figure A1 displays a test case example for the `p_date` class (see Figure 6). Here the test case is written in the EiffelUnit format (http://w3.one.net/~jweirich/software/eiffelunit/), which is a unit-testing framework for Eiffel classes and is part of the XUnit framework family. The test case is encapsulated in a class, and the two parts of the gene (initialization and method calls, see Section 3.2.2)

```
class
        UNIT_TEST_EXAMPLE
inherit
        EUNIT_TESTCASE
feature – Support
  date:P_DATE;

  set_up is
    do
        !!date.make (10)
    end

feature – Tests
  test_comparison is
    local date1:P_DATE
    do
      !!date1;
      date.set(1999,7,5);
      date1.set(1998,7,5);
        assert(date1 < date);
    end

end – UNIT_TEST_EXAMPLE
```

```
class
        UNIT_TEST_EXAMPLE
inherit
        EUNIT_TESTCASE
feature – Support
  date:P_DATE;

  set_up is
    do
        !!date.make (10)
    end

feature – Tests
  test_comparison is
    local date1:P_DATE
    do
      !!date1;
      date.set(1998,7,5);
      date1.set(1998,7,5);
        assert(date1 < date);
    end

end – UNIT_TEST_EXAMPLE
```

Figure A1. Example of a bacterium (or a gene) for the `p_date` class.

```
using System;
namespace Id_1 {
  using System;
  protected class Id_2 {
    [AnAttribute1; AnAttribute2]
    public string aField;

    public ~Id_2() {} //~Id_2

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

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

    [AnAttribute]
    static Type aProperty {
      get {}
      set {
        aVariable = aValue + 3;
        for (int i=0; !Id_6||Id_8!=Id_3; i++)
        {foreach (nodes n in the_tree)
          {anObject.aMethod (param3, param4);}}
      }
    }
    public returnType1 aMethod2 (Type3 param5) {} //aMethod2
  } //Id_2
}
```

```
using System;
namespace Id_1 {
  using System;
  protected class Id_2 {
    [AnAttribute1; AnAttribute2]
    public string aField;

    public ~Id_2() {} //~Id_2

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

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

    [AnAttribute]
    static Type aProperty {
      get {}
      set {
        aVariable = aValue + 3;
        for (int i=0; !Id_6||Id_8!=Id_3; i++)
        {while(cond1){
          aVariable1++;}}
      }
    }
    public returnType1 aMethod2 (Type3 param5) {} //aMethod2
  } //Id_2
}
```

Figure B1. Example of a bacterium (or a gene) for the C# parser.

are written in two separate methods. The initialization part is done by the `set_up` method, and method calls are in the `test_comparison` method. The framework always calls the `set_up` method before executing the test method.

The figure also displays an example of mutation operator application. Here the method call `date.set` in the test case on the left has been chosen for mutation. A new gene has been created on the right of the figure, and the parameters of the `date.set` method have been changed from (1999,7,5) to (1998,7,5).

APPENDIX B. EXAMPLE FOR C#

Figure B1 gives an example of bacterium (or gene) written in C#. This is an example of a C# source file that can be passed as an input to the C# parser. This file contains 20 nodes from the syntax tree (C# constructs). The figure also illustrates the mutation operator. The bold `foreach` node in the left-hand 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-hand source file).

ACKNOWLEDGEMENT

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

REFERENCES

1. Baudry B, Fleurey F, Le Traon Y, Jézéquel J-M. Genes and bacteria for automatic test cases optimization in the .NET environment. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '02)*, Annapolis, MD, November 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 195–206.
2. Beck K, Gamma E. Test-infected: Programmers love writing tests. *Java Report* 1998; **3**(7):37–50.
3. DeMillo R, Lipton R, Sayward F. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 1978; **11**(4):34–41.
4. Voas JM, Miller K. The revealing power of a test case. *Software Testing, Verification and Reliability* 1992; **2**(1):25–42.
5. Kim S-W, Clark JA, McDermid JA. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability* 2001; **11**(4):207–225.
6. Baudry B, Le Traon Y, Jézéquel J-M, Hanh VL. Trustable components: Yet another mutation-based approach. *Proceedings of the 1st Symposium on Mutation Testing*, San Jose, CA, October 2000, Wong WE (ed.). Kluwer Academic: Dordrecht, 2000; 69–76.
7. Moore I. Jester—a JUnit test tester. *Proceedings of XP '2001*, Villasimius, Sardinia 2001; 84–87.
8. MSDN. .NET homepage. http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000519 [March 2002].
9. MSDN. C# introduction and overview. http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpintro.asp [March 2002].
10. Offutt AJ, Pan J, Tewary K, Zhang T. An experimental evaluation of data flow and mutation testing. *Software—Practice and Experience* 1996; **26**(2):165–176.
11. Meyer B. *Object-oriented Software Construction*. Prentice-Hall: Englewood Cliffs, NJ, 1992.
12. DeMillo R, Offutt AJ. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 1991; **17**(9):900–910.
13. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology* 1992; **1**(1):5–20.

14. Deveaux D, Jézéquel J-M, Le Traon Y. Self-testable components: From pragmatic tests to a design-for-testability methodology. *Proceedings of TOOLS '99 (Technology of Object Oriented Languages and Systems)*, Nancy, France, June 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 96–107.
15. Goldberg DE. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley: Reading, MA, 1989.
16. Holland JH. *Adaptation in Natural and Artificial Systems*. University of Michigan Press: Ann Arbor, MI, 1974.
17. Rosenzweig ML. *Species Diversity in Space and Time*. Cambridge University Press: Cambridge, U.K., 1995.
18. Baudry B, Fleurey F, Le Traon Y, Jézéquel J-M. Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components. *Proceedings of ASE '02 (Automated Software Engineering)*, Edinburgh, U.K., September 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 253–256.
19. Ma Y-S, Kwon Y-R, Offutt AJ. Inter-class mutation operators. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '02)*, Annapolis, MD, November 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 352–363.
20. Chevalley P. Applying mutation analysis for object-oriented programs using a reflective approach. *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, Macao, China, December 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 267–270.
21. Alexander RT, Offutt AJ, Bieman JM. Fault detection capabilities of coupling-based OO testing. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '02)*, Annapolis, MD, November 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 207–218.
22. Ghosh S, Mathur A. Interface mutation. *Software Testing, Verification and Reliability* 2001; **11**(4):227–247.
23. Yoon H, Choi B. Effective test case selection for component customization and its application to Enterprise JavaBeans. *Software Testing, Verification and Reliability* 2004; **14**(1):227–247.
24. Michael CC, McGraw G, Schatz MA. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 2001; **27**(12):1085–1110.
25. Pargas R, Harrold MJ, Peck R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability* 1999; **9**(4):263–283.
26. Jones BF, Sthamer H-H, Eyres DE. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 1996; **11**(5):299–306.

