feature

testing

Automatic Test Case Optimization: A Bacteriologic Algorithm

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

Finding the best test cases for software component testing can be expensive and labor intensive. A bacteriological algorithm can automatically generate and optimize test cases.

Because testing is extensively used to assess a software product's quality, assessing the quality of the testing itself is important. Indeed, the more efficient the test cases are, the more testing we can perform in a given time and therefore the more confidence we can have in the software. One approach to building confidence in test cases is *mutation analysis*,[1] which introduces faults in the software under test. We assume that test cases are good if they detect these faults. This approach, which has been successfully applied to qualify unit test cases for object-oriented classes,[2,3] gives programmers useful feedback on the "fault-revealing power" of their test cases. It also offers an estimate of how many new test cases they need to better test a given software component.

While generating a set of basic test cases might be easy, improving the set's quality usually requires prohibitive effort. Indeed, the test cases that testers generally provide easily cover 50–70 percent of the introduced faults, but improving this score to 90–100 percent is time consuming and therefore expensive. So, automating the test optimization process could be extremely helpful.

Improving test cases automatically is a nonlinear optimization problem. To solve this problem, we've developed a *bacteriologic algorithm*, adapted from genetic algorithms,[4] that can generate and optimize a set of test cases. A .NET component that parses C# source files[5] illustrates our algorithm.

The C# parser

The parser, which is implemented in C#, takes a set of C# source files as an input and builds the corresponding syntax tree. Figure 1 shows the UML class diagram for this parser. This system has 32 classes that we divide into three main parts. First, `CSNodeBuilder` is the main class for building the syntax tree. Second, the inheritance hierarchy under `CSNode` corresponds to the node types in the C# abstract syntax tree. The third part is the `NodeVisitor` interface and its different implementations. These classes correspond to the application of the Visitor design pattern,[6] which enables implementing different processing on the syntax tree.

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

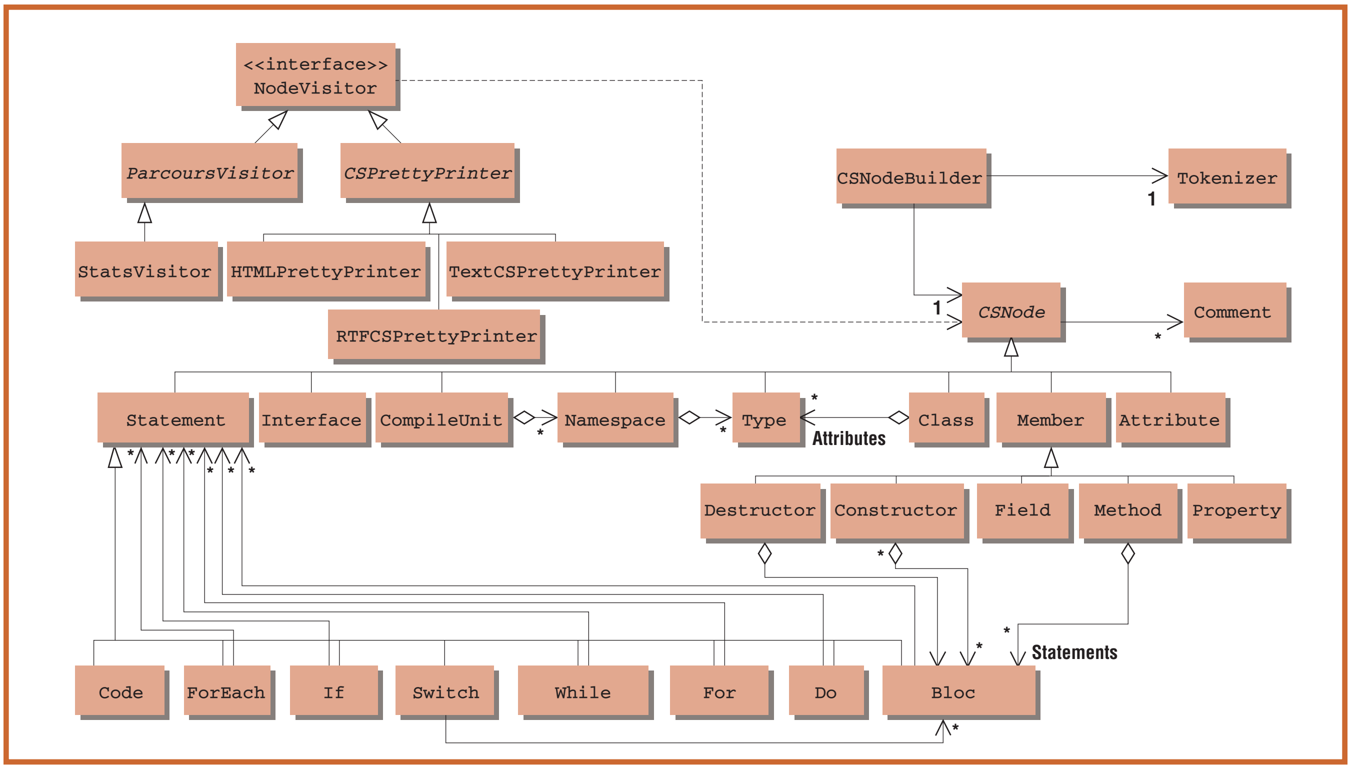


Figure 1. A UML class diagram for the C# parser.

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

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

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

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

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

Figure 2. An example test case for the C# parser.

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

Figure 3. The `accept` method.

Mutation analysis

Mutation analysis was first designed to create effective test data with important fault-revealing power.[7] It introduces faults in the component under test (CUT) to create a set of *mutants*, each containing a fault. The goal is then to design a set of test cases that distinguishes the component from its mutants. A mutant that a test case has detected is said to be *killed* by the test case; otherwise it's *alive*. When generating mutants, you might create *equivalent mutants*; that is, no test case can distinguish the mutant's output from the original component's output.

You can evaluate the quality of a set of test cases by its *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 set T is

$$MS(T) = 100(d/(m - equiv))$$

In practice, faults are modeled by a set of *mutation operators*, each operator representing a class of software faults.[3,8] Here, we apply mutation analysis on a component built with several classes. Because the number of mutants increases with the component's size, and the execution time increases with the number of mutants (all test cases must be executed against all the mutants), we must choose a limited number of mutation operators:

- The LOR operator replaces each occurrence of a logical operator (AND, OR, NAND, NOR, or XOR) with a different operator; in addition, a Boolean expression can be replaced by TRUE or FALSE.
- The NOR operator suppresses a statement or a block of statements.

For example, for the `accept` method in Figure 3, we could create a LOR mutant by replacing statement 1 with

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

To create a NOR mutant, we could delete statement 2.

Assuming that all classes in the component have been tested at the unit level, we believe that two operators (LOR and NOR) are sufficient. This belief is realistic because after unit testing, the testing focuses on the interactions between units. These operators guarantee code and predicate coverage, which is sufficient for testing the interactions.

When a set of mutant components is automatically generated with the selected mutation operators, the test cases are executed against each mutant. If executing a test case on a mutant produces an output different from that of executing it on the initial program, the test case kills the mutant. This specific oracle function is meaningful for evaluating the test case's quality. Indeed, as we mentioned earlier, mutation analysis aims to check the test cases' ability to detect the errors that have been intentionally injected in the initial program. Thus, it aims to check if the test cases can detect the difference between the initial program and the mutant. Once mutation analysis has generated good test cases, they're executed against the initial program to detect real errors.

If no test case kills a mutant program, a diagnosis step determines why. The mutant might be alive because the test cases are too weak or because it's an equivalent mutant. Our algorithm automates test case optimization after this step.

A bacteriologic algorithm for automatic test optimization

We call our algorithm "bacteriologic" because it's inspired by *evolutionary ecology* and, more particularly, *bacteriologic adaptation*. Evolutionary ecology is the study of living organisms in the context of their environment, with the aim of discovering how they adapt.[9] Its basic concept is that in a heterogeneous environment, you can't find one individual that fits the whole environment. So, you need to reason at the population level. This matches the intuition for the problem we want to solve: you can't generate a single perfect test case to kill all mutants; instead, you need to generate and improve a global set of test cases.

The bacteriologic algorithm takes as input an initial set of test cases, and it outputs a good set of test cases. The algorithm evolves incrementally (each increment is called a *generation*) and consists of a series of mutations on test cases, to explore the scope of solutions. The algorithm builds the final set incrementally by memorizing test cases that can improve the set's quality (a *fitness function* evaluates this quality). As the execution unfolds, there are two test sets: the *solution set* that the algorithm is building and the *bacteriologic medium*, a set of potentially interesting test cases. Several stopping criteria can exist for the global process: after a number of generations, when the solution set reaches a minimum fitness value, if the set's fitness value hasn't changed for a number of generations, and so on.

We denote the program's input domain as $\mathcal{TC}$. Each generation involves four basic functions. The *fitness function* (*fitness*: $2^{\mathcal{TC}} \rightarrow \mathbb{R}^+$) computes a real number that evaluates the quality of a set of test cases regarding the global objective. In the case of automatic test generation, this function can be based on the control graph's coverage rate, the mutation score, or any other test adequacy criterion. We also define a fitness function for a single test case. It's called *relFitness* (*relFitness*: $\mathcal{TC} \times 2^{\mathcal{TC}} \rightarrow \mathbb{R}^+$) and computes the fitness of a test case *tc* (relatively to the fitness of a set of test cases *TCS*) as follows:

```
relFitness(TCS, tc)
= fitness(TCS ∪ {tc})
  – fitness(TCS).
```

The *memorization function* (*mem*: $\mathcal{TC} \rightarrow$ Boolean) takes a test case as an input and determines its relative fitness. If the fitness satisfies a given condition (for example, if it exceeds a given threshold), the function returns TRUE, and the algorithm memorizes the test case.

The *mutation function* (*mutate*: $\mathcal{TC} \rightarrow \mathcal{TC}$) generates a new test case by slightly altering an ancestor test case. This operator is crucial for the algorithm because it's the one that creates new information in the process. By recursive applications of this operator, we should explore the whole set of possible test cases $\mathcal{TC}$.

Finally, the *filtering function* (*filter*: $2^{\mathcal{TC}} \rightarrow 2^{\mathcal{TC}}$) aims to periodically delete useless test cases from the bacteriologic medium to control the memory space during execution.

In addition to these four functions, the algorithm requires that we set two parameters:

- The *memorization threshold*. This limits the number of memorized test cases.
- The *size of the test cases*. If the grammar for test cases is available, the size is the number of nodes in the syntax tree.

The algorithm manipulates only test cases of the same size. This might appear as a limitation, but it's necessary. If the mutation function can make test cases grow to improve their fitness, the size of memorized test cases will always grow. Indeed, a bigger test case is always more fitted than a smaller one (this seems obvious intuitively and has been experimentally verified). However, the bigger a test case is, the longer it takes to execute. On the other hand, if test cases are too small, either they can't kill enough mutants or they kill so few that we need a very large set of them to reach a good mutation score. So, it's important to have a fixed size that we tune before we run the algorithm.

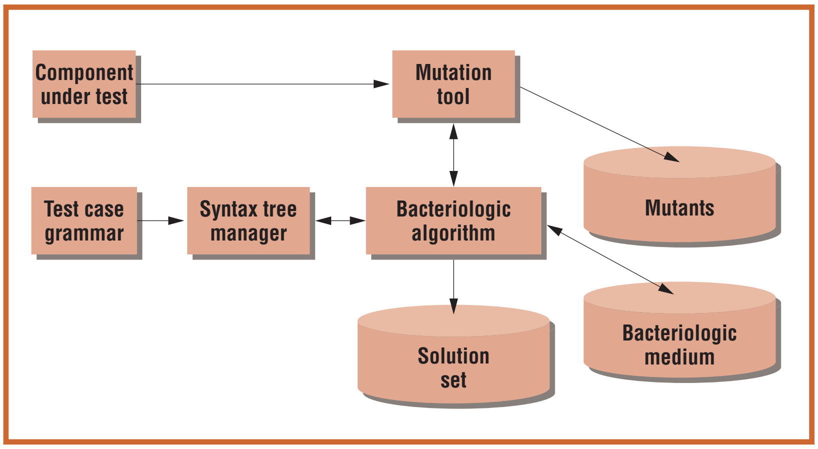


Figure 4. A framework for test case optimization with a bacteriologic algorithm.

Running the bacteriologic algorithm

Figure 4 displays the global architecture we used to automatically generate a set of test cases for the parser. However, this architecture is generic enough to be adapted to any problem that consists of improving the mutation score of test cases where a grammar can describe their structure. It has three main components: the bacteriologic algorithm, the *mutation tool*, and the *syntax tree manager* (STM). The process takes two input data: the CUT and the grammar describing test cases for the CUT. The output is a set of test cases with a high mutation score.

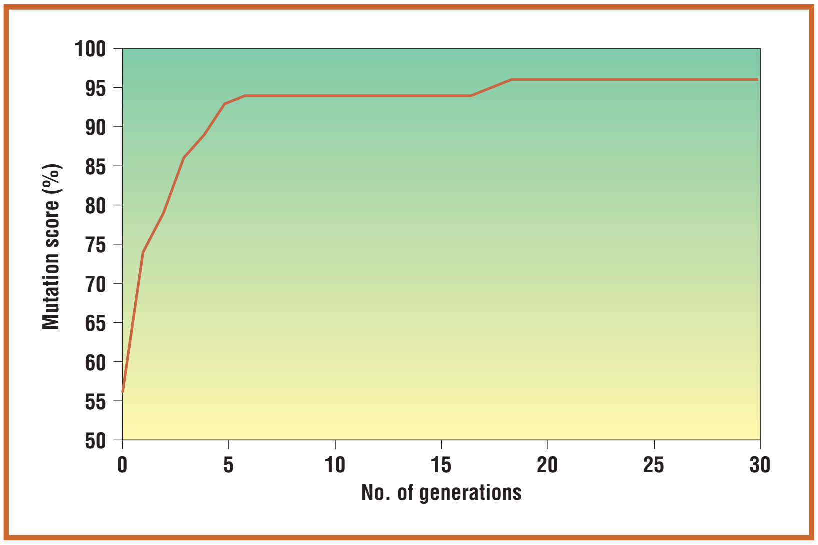


Figure 5. The results of a bacteriologic approach for system test data optimization.

At the architecture's center is the bacteriologic algorithm, which we described earlier. It manipulates two data sets: the bacteriologic medium and the solution set. The mutation tool computes test case fitness. As we explain later, we use the STM for the bacteriologic algorithm's initialization and for its mutation function.

Initialization

The initial set of test cases can either be written by hand or automatically generated with a random generator. For our experiments with the C# parser, the STM randomly generated the initial set from the C# grammar.

We conducted several experiments to tune the size of the test cases,[10] which we set at 25 nodes. We passed this size as a parameter to the STM to generate the initial test cases.

Fitness function

We use the mutation score of a set of test cases as that set's fitness function. To compute this function, we developed the NMutator mutation tool, which automatically generates all NOR mutants. NMutator parses C# components to find all possible locations in the code where it can introduce an error. Then it generates all corresponding mutant components.

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

Memorization function

This function computes the relative fitness of all test cases in the bacteriologic medium. In our case, this is a test case's mutation score relative to the solution set's mutation score. This relative fitness thus represents the proportion of mutants a test case *tc* can kill that the test cases in the solution set haven't killed. The *relative mutation score* of a set of test cases is

$$relMS(TCS, tc) = MS(TCS \cup \{tc\}) - MS(TCS),$$

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

Once the memorization function has computed relative mutation scores for all test cases in the bacteriologic medium, it selects the test cases whose relative mutation score exceeds the memorization threshold (which is a global parameter of the algorithm).

Mutation function

This function randomly selects test cases in the bacteriologic medium. The random selection is weighted by the test cases' relative fitness (better test cases have higher chances to be selected). The selected test cases are then mutated to create new test cases that are added to the bacteriologic medium for the next generation. The test cases can be represented by the abstract syntax tree representing the program, and mutating a test case consists of replacing one node in the tree by another licit node. (By licit node, we mean that the node replacement must build a syntactically correct test case.) Because the STM can access the test cases' grammar, it can parse a source test case, select a node in the tree, and find a licit node to build a target test case. The STM thus handles the bacteriologic algorithm's mutation function.

For example, in the test case in Figure 2, the `foreach` node (lines 18 and 19) can be replaced with a `while` node such as this one:

```
while(cond1){aVariable1++;}}
```

Filtering function

Our algorithm uses two different implementations of this function to delete test cases from the bacteriologic medium:

- Delete any test case whose relative mutation score is equal to 0 (the function kills no mutant that the test cases in the solution set haven't killed).
- Reduce the coverage matrix by deleting redundant test cases. For example, some test cases might kill the same mutants. Keeping all of them is useless.

Besides these two implementations, we could use many other techniques for minimizing or prioritizing sets of test cases.[11]

Results

Figure 5 shows the results of executing the bacteriologic algorithm for the C# parser. We aimed to generate a set of test cases that could kill the 500 mutants generated for the parser. The initial set consisted of 30 test cases. The best case had a 57 percent mutation score and was memorized at the first generation (the solution set's initial score). As we mentioned before, we set the test case size at 25 nodes, and the memorization threshold was 20 percent. After 30 generations, the algorithm generated seven new test cases, and the final set had a mutation score of 96 percent. The generated test cases let us actually detect errors in the parser. After fixing these errors, we ran the bacteriologic algorithm again (changing the component changes the set of mutants, and running another mutation analysis with these new mutants is necessary). With such an incremental process, we could establish good confidence in both the set of test cases and the component.

Because the bacteriologic algorithm is pseudorandom, the results for the same set of mutants varied slightly with each execution. For example, the number of generated test cases ranged from seven to 10 throughout our experiments.

Because genetic algorithms have often been used for automatic test case generation, we compared our bacteriologic algorithm with a genetic algorithm.[4] Each algorithm executed 50 times. Here are the basic results:

- The genetic algorithm ran 200 generations for an average mutation score of 85 percent (ranging from 80 to 87 percent). Each run required executing an average of 480,000 test cases.
- The bacteriologic algorithm ran 30 generations for an average mutation score of 96 percent (ranging from 92 to 97 percent). Each run required executing an average of 46,375 test cases.

Judging by the number of generations needed to reach the best score, the bacteriologic approach appears to converge more quickly: 30 instead of 200. However, because each algorithm performs a different computation to go from one generation to the next, we provide more comparable figures: the number of times a mutant program executed. This is a better estimation of the complexity because executing a mutant is equally time consuming in both approaches.

In addition, the bacteriologic algorithm is easier to tune. This makes it more reusable for test generation and optimization problems. Removing parameters also makes the model more controllable because the algorithm's execution exhibits less randomness. The approach is thus more stable than a genetic one.

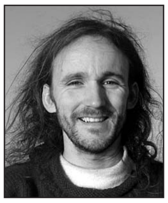
The bacteriologic algorithm could also be helpful for complex positioning problems or data mining.

Computer science has often been inspired by biological processes: neural networks, genetic algorithms, and so on. Following this tradition, this work was inspired by bacteriologic adaptation: a rapid-evolution phenomenon that can adapt bacteria to a large and changing environment. As we showed, we've successfully applied the proposed bacteriologic algorithm for automatic test generation. In a broader way, we strongly believe it could be helpful in many other fields, such as complex positioning problems (antennas for cell phones, urban planning, and so on) or data mining.

Acknowledgments

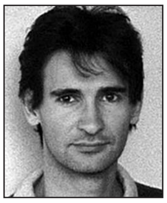
We're grateful to Stan Rifkin for his very helpful comments on an earlier version of this article. Many thanks also to Françoise Burel, director of Rennes I University's Ecosystem Functioning and Conservation Biology lab, for her helpful suggestions in defining the bacteriologic algorithm.

About the Authors



Benoit Baudry is a junior researcher in software engineering at INRIA, on the Triskell project team. His research interests concern software testing, fault localization, design for testability, and software modeling in the context of model-driven software development. He received his PhD in computer science from the University of Rennes. Contact him at Irisa, Université de Rennes 1, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France; benoit.baudry@irisa.fr.

Franck Fleurey is a PhD student on the Triskell project team at the University of Rennes. His research interests are model-driven engineering and software validation. Contact him at Irisa, Université de Rennes 1, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France; franck.fleurey@irisa.fr.



Jean-Marc Jézéquel is a professor at the University of Rennes, where he leads the Triskell project team. His interests include model-driven software engineering based on object-oriented technologies for telecommunications and distributed systems. He's the author of *Object-Oriented Software Engineering with Eiffel* (Addison-Wesley, 1996) and *Design Patterns and Contracts* (Addison-Wesley, 1999). He's a member of the steering committee of the MoDELS/UML conference series. He also serves on the editorial boards of *IEEE Transactions on Software Engineering*, the *Journal on Software and System Modeling*, and the *Journal of Object Technology*. He received his PhD in computer science from the University of Rennes. Contact him at Irisa, Université de Rennes 1, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France; jean-marc.jezequel@irisa.fr.

Yves Le Traon is a research engineer at France Télécom R&D and an associate member of Irisa. His research interests include object-oriented testing, design for testability, and software measurement. He received his PhD in computer science from the Institut National Polytechnique de Grenoble. Contact him at Irisa, Université de Rennes 1, Campus Univ. de Beaulieu, 35042 Rennes Cedex, France; yves.le_traon@irisa.fr.

References

1. R. DeMillo, R. Lipton, and F. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” *Computer*, vol. 11, no. 4, 1978, pp. 34–41.
2. B. Baudry et al., “Trustable Components: Yet Another Mutation-Based Approach,” *Proc. 1st Symp. Mutation Testing*, Kluwer Academic Publishers, 2000, pp. 69–76.
3. S.-W. Kim, J.A. Clark, and J.A. McDermid, “Investigating the Effectiveness of Object-Oriented Testing Strategies Using the Mutation Method,” *Software Testing, Verification and Reliability*, vol. 11, no. 4, 2001, pp. 207–225.
4. B. Baudry et al., “Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment,” *Proc. Int'l Symp. Software Reliability Eng.* (ISSRE 02), IEEE CS Press, 2002, pp. 195–206.
5. F. Fleurey, “C# Pretty Printer Home Page,” http://franck.fleurey.free.fr/CSPrettyPrinter/index.htm.
6. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
7. A.J. Offutt et al., “An Experimental Evaluation of Data Flow and Mutation Testing,” *Software: Practice and Experience*, vol. 26, no. 2, 1996, pp. 165–176.
8. A.J. Offutt et al., “An Experimental Determination of Sufficient Mutant Operators,” *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, 1996, pp. 99–118.
9. E.R. Pianka, *Evolutionary Ecology*, Addison-Wesley, 1999.
10. B. Baudry et al., “Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components,” *Proc. Automated Software Eng.* (ASE 02), IEEE CS Press, 2002, pp. 253–256.
11. G. Rothermel et al., “Prioritizing Test Cases for Regression Testing,” *IEEE Trans. Software Eng.*, vol. 27, no. 10, 2001, pp. 929–948.