GPTesT: A Testing Tool Based on Genetic Programming

Maria Cláudia Figueiredo Pereira Emer
UFPR - Curitiba CP: 19081,
81531-970, Brazil
mpereira@inf.ufpr.br

Silvia Regina Vergilio
UFPR - Curitiba CP: 19081,
81531-970, Brazil
silvia@inf.ufpr.br

Abstract

Genetic Programming (GP) has recently been applied to solve problems in several areas. It has the goal of inducing programs from test cases by using the concepts of Darwin's evolution theory. On the other hand, software testing, that is a fundamental and expensive activity for software quality assurance, has the objective of generating test cases from the program being tested. In this sense, a symmetry between induction of programs based on GP and testing is noticed. Based on such symmetry, this work presents GPTesT, a testing tool based on GP. Fault-based testing criteria generally derive test data using a set of mutant operators to produce alternatives that differ from the program under testing by a simple modification. GPtesT uses a set of alternatives genetically derived, which allow the test of interactions between faults. GPTesT implements two test procedures respectively for guiding the selection and evaluation of test data sets. Examples with these procedures show that the approach can be used as a testing criterion.

1 INTRODUCTION

The use of software products in most areas of human activities has generated a growing interest in software quality assurance. Software Engineering techniques and tools were proposed with the goal of increasing the quality of the software being developed. In this context, the software testing activity has gained importance during the last decade and is considered fundamental.

In the literature, there are three groups of testing techniques proposed to reveal a great number of faults with minimal effort and costs: 1) functional technique: uses functional specification of a program to derive test cases; 2) structural technique: derives test cases based on paths in the control flow graph of the program; 3) fault-based technique: derives test cases to show the presence or absence of typical faults in a program.

These techniques are generally associated to the testing criteria. A criterion is a predicate to be satisfied to consider the testing activity ended, that is, to consider a program tested enough [16, 25]. It helps the tester in two major tasks: test case selection and test case evaluation.

Fault-based criteria consider that most programmers do their programs very similar to the correct program, according to a specification. This fact is known as "competent programmer hypothesis [8]". When the users test a program, they use the correct program that they have in mind, and if the program P being tested is not correct, there is a set of alternatives for P that can include at least one correct program. The fault-based criteria explore the use of alternatives for testing P [8, 9, 14, 20]. In most cases, the alternative program differs from P by a simple syntactic modification, that is, only a fault at a time is introduced. They assume that complex faults are detected by analyzing simpler faults, this assumption is named "coupling effect" [8, 22].

Some works [9, 14] assume only necessary conditions for discovering faults; that is, to reveal a fault is necessary to produce only an intermediate different state in the program and in its alternative, after the modified statement. This is assumed because determining sufficient conditions, which are the conditions to produce different final states, is undecidable (task related to the term coincidental correctness [3]). However, Morell[20] points out that these assumptions ignore the global effect of faults or interactions of modifications in the program.

Genetic Programming (GP) is a field of the called Evolutionary Computation. The term was popularized by Koza in 1992 [15]. The goal is to use the concepts of Darwin's evolution theory [6] for computer program induction. The concepts are usually applied by genetic operators such as: selection, crossover, mutation and reproduction. During the last decade, GP has received significant attention and been used to solve a large number of problems, mainly in Artificial Intelligence and Engineering Areas [1].

Some authors mention that there is a symmetry between the testing activity and the induction of programs [2, 3, 28]. In this sense, testing is an activity that generates test cases from a program being tested, and GP is a technique that generates programs from test cases.

Based on such symmetry, this work describes GPTesT tool, that supports a GP-based test approach. The alternatives are generated using GP and can differ from P by more than simple modifications. GPTesT guides the tester in two tasks: selection and evaluation of test cases. It allows the test of C programs and uses Chameleon [26], a GP tool. The paper is organized as follows. Section 2 shows aspects related to GP and the Chameleon tool. Section 3 presents a review about the test activity. Section 4 describes GPTesT and Section 5 illustrates the mentioned test procedures. Finally, Section 6 concludes the paper.

2 ABOUT CHAMELEON

Genetic Programming (GP) was introduced by John Koza [15], based on the idea of Genetic Algorithms presented by John Holland [13]. Instead of a population of beings, GP works with a population of computer programs. The goal of the GP algorithm is to naturally select the program that better solves a given problem, through recombination of "genes",. A special heuristic function called fitness is used to guide the algorithm in the process of selecting individuals. This function receives a program and returns a number that shows how close this individual is to the desired solution. First, an initial population of computer programs is randomly generated (Generation 0). After that, the GP algorithm enters a loop that is ideally executed until a desired solution is found.

In this paper, the tool Chameleon [26] illustrates the use of GP for software testing. Chameleon implements a grammar-oriented approach and evolves C programs. It represents the programs using grammar-based derivation trees.

Through the evolution process, genetic operators recombine programs by making modifications directly on their derivation trees. In reproduction, no change is made: the individual is simply replicated to the next generation. It is equivalent to the asexual reproduction of beings. Mutation is the addition of a new segment of code to a randomly selected point of the program.

Crossover is the operator that truly performs recombination of computer programs. This operation takes two parents to generate two offspring. A random point of crossover is selected on each parent and the subtrees below these points are exchanged. It is equivalent to the sexual reproduction of beings. When grammars are used, the crossover operator is restricted and only allows the exchange of tree branches that have been generated using the same production rule.

To execute Chameleon, the user needs to provide the grammar correspondent to the problem to be solved and an initial configuration I of parameters. The parameters are related to the genetic operations as mutation and crossover rates; to the number of runs and size of population; to the derivation tree; and to the name of a file that contains a set T of test cases. These test cases are used to calculate the fitness value of each individual. The number of runs is used to end the process. The individual (or program) with better fitness value is selected. The selection can also be random.

Figure 1 shows an example with the initial configuration I of parameters, including the grammar, for language C, adopted to the problem of calculating the common minimum multiple of two given numbers (*cmm* problem). Chameleon finds, among other, the solution presented in Figure 2.

3 ABOUT TESTING

The main goal of testing is to find an unrevealed fault [21]. Hence, how to derive test cases for revealing as many faults as possible is an important question. This is because it is related to some factors such as efficacy, costs, limitations to automate the testing activity, etc. Other question to be considered is to known whether a program has being tested enough or how to evaluate a data test set T. These two questions, related to generation and evaluation of test cases, are discussed by Rapps and Weyuker in [24].

In order to guide the testing activity and answer the above questions, different testing criteria were proposed. They consider different aspects to derive the test data. Functional criteria use functional specification of a program to derive test cases. Boundary Value Analysis and Cause-Effect Graphs [23] are ex-

```
[begin]
[parameters]
population size=500
number of runs=10
tournament size = 10
maximum depth for initial random programs = 15
maximum depth  during the run = 30
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {%, !=, *, /}
output variable = Z
[result-producing branch productions]
<code>   -> <def> <prog> <result>
<def>    -> float R = 1, A = X, B = Y;
<result> -> Z = (A<op>B) <op> <var>;
<prog>   -> if (<expc>) {<prog1>} else {<atr>}
<prog1>  -> do {<bloco>} while (<expc>);
<bloco>  -> <exp>
<bloco>  -> <bloco> <exp>
<exp>    -> <var> = <var> <opm> <var>;
<exp>    -> <var> = <var>;
<expc>   -> <var> <opc> <cte>
<atr>    -> <var> = <cte>;
<opm>    -> %
<op>     -> *
<op>     -> /
<opc>    -> !=
<var>    -> X
<var>    -> Y
<var>    -> R
<cte>    -> 0
[fitness cases]
source -> cmm.dat
[end]
```

Figure 1: Initial Configuration for Chameleon

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       R=Y;
       Y=X%Y;
       X=R;
     } while (Y!=0)
   }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

Figure 2: A Possible Solution for the cmm Problem

amples of functional criteria. Structural criteria derive test cases based on paths in the control-flow graph of the program. The best known structural criteria are control-flow and data-flow based criteria [16, 24, 27]. Fault-based criteria derive test cases to show the presence or absence of typical faults in a program, considering common errors in the software development process. The best known fault-based criterion is Mutation Analysis [8].

This work focuses on fault-based testing, and the Mutation Analysis criterion will be described in more detail. It consists basically of generating mutant programs for the program P being tested. Mutation Analysis considers two assumptions [8]: 1) the hypothesis of the competent programmer: "Programmers do their programs very similar to the correct program"; 2) coupling effect: "Tests designed to reveal simple faults can also reveal complex faults". It is also based on a set of mutation operators. A mutant is represented by a single mutation in the original program established by a mutation operator.

All mutants are executed using a given input test case set T. If a mutant M presents different results from P, it is said to be dead, otherwise, it is said to be alive. In this case, either there are no test cases in T that are capable to distinguish M from P, or M and P are equivalent. To satisfy the criterion, we have to find a test case set able to kill all non-equivalent mutants; such a test case set T is considered adequate to test P. Then, a mutant will be considered dead if its behavior concerning a test case is different from that of the original program. The Mutation Score, obtained by the relation between the number of mutants killed and the total number of non-equivalent mutants generated, allows the evaluation of the adequacy of the used test case set. The number of equivalent mutants generated is not determined automatically; it is obtained interactively as an entry from the tester, since the equivalence question is, in general, undecidable [5, 8].

In the literature, there are many testing tools. However, the complete automation of testing activity is not possible due to many testing limitations: infeasible paths, equivalent mutants, etc. In general, there is no algorithm to generate a test set that satisfies a given criterion. It is not even possible to determine if such set exists [12]. In spite of these limitations, there are in the literature many works addressing test data generation for satisfying testing criteria. Most recent studies have been exploring Genetic Algorithms [4, 17, 18, 19].

Proteum [10] and Mothra [7] are examples of testing tools based on mutation testing. These tools generate mutants by using mutation operators. Proteum has a set of 71 mutation operators and supports test of C programs. Mothra supports testing of Fortran programs. Different operators are, in general, defined for different programming languages and the mutants differ from the program being tested by a simple modification. Morell[20], however, points out that such fact ignores the global effect of faults or interactions of modifications in the program.

This work proposes the use of GP to derive the mutants. This can produce alternatives that are very different from the original program and consequently can test global effects of faults. These aspects are discussed in the following section.

4 GPTesT

In this section we describe GPTesT (**G**enetic **P**rogramming-based **Tes**ting **T**ool) implemented to support GP based testing. It implements two test procedures for selection and evaluation of test cases, showing that the approach can be used in the same way as a testing criterion, such as Mutation Analysis.

Figure 3 contains the Use Case Diagram for GPTesT. Next, we present a brief description and purpose of each use case and describe the main GPTesT functionalities.

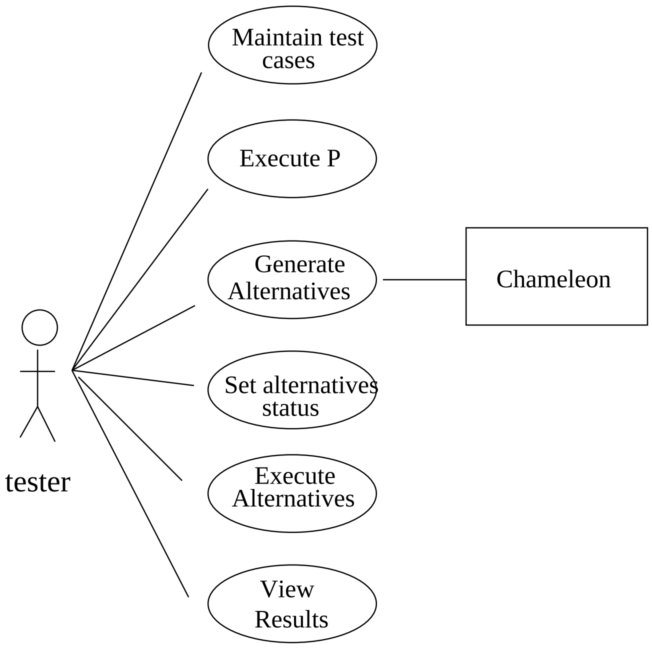

Figure 3: GPTesT: Use Case Diagram

- Maintain test cases: this use case is related to different functions for test case maintenance. The tester can add a new test case, delete or disable an existent, as well, visualize the obtained output after execution of the program P being tested. GPTesT saves all the given test cases as part of the testing session for P.

- Execute P: this use case executes P with all the non-executed test cases and saves the obtained output. The tester analyzes the output. If the output is different from the expected, a failure occurred. In this case, the tester must correct P and start a new testing session.

- Generate alternatives using Chameleon tool: this use case runs the Chameleon tool with the configuration I, as illustrated in Section 2. The tester gives I as entry. GPTesT selects the programs generated by Chameleon to obtain the set A of alternatives. This selection discards some anomalous and equivalent programs that can be syntactically determined.

- Set alternative status: each alternative has a status. This status can be:

  - anomalous: the alternative has an anomalous behavior such as division by zero, loop forever, etc.
  - equivalent: the alternative computes the same function of P, producing the same outputs that P produces for any input.
  - dead: the alternative has already produced a different output for a test case when compared with P.
  - alive: the alternative has produced the same output produced by P for all enabled test cases.

  After executing the alternatives, GPTesT automatically updates the alternative status. However, the tester has to identify the equivalence of programs and to set the status of an equivalent alternative. As mentioned in Section 3, there is no algorithm to determine whether two programs compute the same function. This is an undecidable question and all fault-based testing tools have this limitation.

- Execute the alternatives: this use case executes all the alive alternatives from A with all the non-executed test cases.

- View the results: this use case allows the tester to visualize the alternatives and their status, the test cases and a testing score, similarly to other fault-based testing tools and Proteum. To calculate the score, GPTesT uses the following formula:

$$S_M(P,T) = \frac{A_d(P,T)}{A(P) - A_e(P)}$$

where:

- P: program being tested;
- T: a test data set;
- $S_M(P,T)$: the coverage score;
- $A_d(P,T)$: total number of dead alternatives;
- $A(P)$: total number of alternatives;
- $A_e(P)$: total number of equivalent alternatives.

This initial version of GPTesT allows the unit test of programs in C language. GPTesT, as well Chameleon, are oriented to C functions, where only a C function is tested at each time. All the results are saved in files, which are in a directory. To generate the executable alternatives, GPTesT uses the compilation command from I (Chameleon configuration).

GPTesT was developed using the Unified Modeling Language (UML) and implemented in C++. Figure 4 presents the main class diagram for GPTesT. More details about GPTesT implementation are in [11].

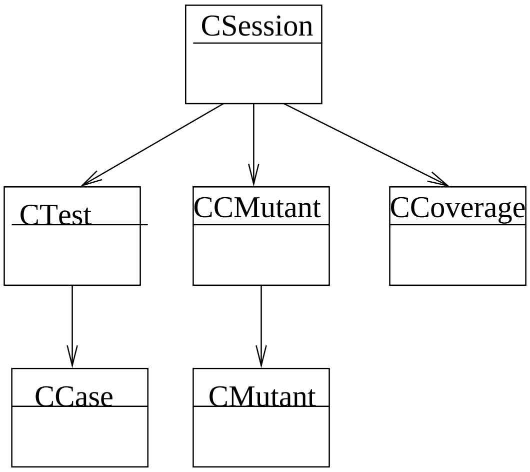

Figure 4: GPTesT: Main Classes

According to some authors [16, 24], a testing criterion or tool must support two testing procedures: selection and evaluation of test cases. Next section illustrates these procedures using GPTesT.

5 TEST PROCEDURES USING GPTEST

5.1 Selection of Test Cases

To illustrate each step of the selection procedure, we use the *cmm* program, whose source code is in Figure 5. This program prints the common minimum multiple of two given numbers.

Suppose that the tester wants to test *cmm*, and does not have any test case. GPTesT guides the tester in the task of test case selection, using GP to perform a fault-based testing. The tester takes the following steps:

```
int cmm (int a, int b)
{
    int  A, B, r;
    A = a;
    B = b;
    if (b!=0)
      do {
        r = A%B;
        A = B;
        B = r;
      } while (r!=0);
    else
      a=0;
    return (a*b)/A;
}
```

Figure 5: Source Code of *cmm* program

1. GPTesT initialization: gives initial information for GPTesT summarized in Table 1.

Table 1: Initial Information to GPTesT

Section Name	cmm
Source Code	cmm.cpp
Initial Chameleon Configuration I	illustrated in Section 2

2. Selection of alternatives: for *cmm* a set of 44 alternatives were generated. Examples of these alternatives are in Figure 6.
3. Generation of test cases to kill the alternatives: to kill the alternative, the tester has to identify a test case that produces an output that differs from P output. Observe that the test case (a=2, b=4) kills the alternative from Figure 6b. P produces 4 and the alternative produces a division by zero.

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (X!=0){
     do {
       R=R%R;
       R=Y;
       X=X%Y;
       Y=X;
       X=R;
     } while (Y!=0)
   }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

a)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       Y=X%Y;
       X=Y
       R=X%R;
      } while (Y!=0)
    }
    else {
      Y=0;
    }
    return (A*B)/R;
 }
```

b)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       R=Y;
       Y=X%Y;
       X=R;
     } while (Y!=0)
    }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

c)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (R!=0){
     do {
       Y=X%Y;
       R=Y%R;
     } while (Y!=0)
    }
    else {
      X=0;
    }
    return (A*B)/R;
 }
```

d)

Figure 6: Examples of generated alternatives

4. Execution of the programs: using the compilation command of I and the test cases given by the tester, GPTesT executes P and the set of alternatives, producing results shown in Figure 7. The results show how many alternatives are dead, alive or equivalent and the score calculated. This final score was obtained with a set of 6 test cases.

5. Addition of new test cases: now, the tester visualizes the alive alternatives and continues the generation of test cases, repeating Steps 3 and 4 until all the non-equivalent alternatives are dead or the desired score is obtained. During this step, the tester manually identifies the equivalent alternatives. Figures 6c shows an example of equivalent alternative, that is identified by the tester.

Figure 8 presents the final status obtained for *cmm*. A score equal to 1 shows that all non-equivalent alternatives are dead using the test cases.

Total Number of Alternatives: 44 Anomalous Alternatives: 0 Live Alternatives: 3 Equivalent Alternatives: 0 Number of Test Cases: 6 Coverage Score: 0.931818

Figure 7: GPTesT Results

Total Number of Alternatives: 44 Anomalous Alternatives: 0 Live Alternatives: 0 Equivalent Alternatives: 3 Number of Test Cases: 6 Coverage Score: 1

Figure 8: Final status for program *cmm*

5.2 Evaluation of a test set

The tester also uses GPTesT for evaluation of a test set T. Consider the program P, which prints the greatest of its three inputs. There is a test set T for P, presented in Table 2. The tester desires to evaluate how good T is. GPTesT helps it in this task. The tester must follow the evaluation procedure described next. Observe that its two first steps are the same steps as the selection procedure.

1. GPTesT initialization.
2. Generation of alternatives.
3. Addition of all test cases from T.
4. Execution of P and of the alternatives using the available test cases.
5. Determination of equivalent alternatives.
6. Analysis of the score for T. The final results for T is in Figure 9

According to the tester's goals, T can be considered good "enough" and the testing activity ends. The evaluation procedure is also used to compare two test cases sets. For example, we can consider that the greater the score the better the set.

Test Case: 1)
Dead Alternatives: 9
Test Case: 2)
Dead Alternatives: 99
Test Case: 3)
Dead Alternatives: 2
Test Case: 4)
Dead Alternatives: 0
Test Case: 5)
Dead Alternatives: 0
Test Case: 6)
Dead Alternatives: 2
Execution Time: 00:01:08h

Total Number of Alternatives: 127
Anomalous Alternatives: 0
Live Alternatives: 15
Equivalent Alternatives: 0
Number of Test Cases: 6
Coverage Score: 0.88189

Figure 9: Status obtained for the test set T

When incorrect outputs of P are obtained, we have to remove the fault and continue the procedure being conducted. When we test a program, we usually follow the two procedures. We can perform the evaluation procedure with a functionally or randomly derived test set T and after this, we start the selection procedure on Step 3, to get the desired score.

Table 2: Test Case Set T

Number	a	b	c
1)	0	1	2
2)	1	2	0
3)	1	0	2
4)	4	5	6
5)	5	6	4
6)	6	4	5

6 CONCLUSIONS

This work presented a framework, named GPTesT, to support the use of Genetic Programming (GP) in the software testing activity. GPTesT allows the use of a new approach to fault-based testing.

The traditional approaches and tools are usually based on mutation operators. An operator is used to generate an alternative program that differs from the program under testing by a simple modification. GPTesT permits the alternative selection by using Chameleon, a GP-based tool. The alternatives do not necessarily differ from the original program by only one modification, and this permits to test interactions among faults, and to reveal other kind of faults than those reveled by the mutation operator approach.

The code of the program under testing is not used to derive the alternatives. This is an advantage during the maintenance phase. All alternatives continue valid. The user decides whether other alternatives will be generated. For the operator mutation approach and structural testing criteria, all the required elements must be generated again since they use the code to establish the testing requirements.

This work presents examples, showing that GPTesT supports two test procedures: test data set evaluation and selection. These procedures are a basic requirement, supported by most testing and criteria tools.

In spite of GPTesT helps the test of C programs and interacts with Chameleon, the GP approach implemented by GPTesT is independent on the used language. GPTesT implementation also permits future extensions. A possible extension is to generate alternatives using other GP tools that evolve programs written in other languages or paradigms. We intend to extend GPTesT to deal with Lisp programs.

Similar to other testing tools found in literature, GPTesT has some limitations. This happens due to the undecidibility related to the equivalence between programs and to the generation of test cases. However, in a future work we will extend GPTesT with mechanisms to reduce these limitations. The mechanisms are heuristics to determine equivalent alternatives and genetic algorithms to automatically generate test cases, helping the tester during the procedures exemplified in this paper.

References

[1] *Proceedings of Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2000.

[2] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, 1995.

[3] T. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, Vol. 18(1):31–45, November 1982.

[4] I. Chung. Automatic testing generation for mutation testing using genetic operators. In *Proceedings of SEKE*. San Francisco, June 1998.

[5] W. Craft. *Detecting Equivalents Mutants Using Compiler Optimization*. Master's Thesis, Department of Computer Science, Clemson University, Clemson-SC, 1989.

[6] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. 1859.

[7] R. De Millo, D. Gwind, and K. King. An extended overview of the mothra software testing environment. In *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151. Computer Science Press, Banff - Canada, July 19-21 1988.

[8] R. De Millo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. C-11:34–41, April 1978.

[9] R. De Millo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, Vol. SE-17(9):900–910, September 1991.

[10] M. E. Delamaro and J. Maldonado. A tool for the assesment for test adequacy for c programs. In *Proceedings of the Conference on Performability in Computing Systems*, pages 79–95. East Brunswick, New Jersey, USA, July 1996.

[11] M. Emer. *Seleção e Avaliação de Dados de Teste Baseadas em Programação Genética*. Master's Thesis, DInf - UFPR, Curitiba-PR, March 2002. (in Portuguese).

[12] F. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., October 1987.

[13] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[14] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, Vol. SE-8(4):371–379, July 1982.

[15] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Slection*. MIT Press, Cambridge, MA, 1992.

[16] J. Maldonado, M. Chaim, and M. Jino. Briding the gap in the presence of infeasible paths: Potential uses testing criteria. In *XII International Conference of the Chilean Science Computer Society*, pages 323–340. Santiago, Chile, October 1992.

[17] G. McGraw and C. Michael. *Automatic Generation of test-cases for software testing*. Technical Report RST Corporation, 1997.

[18] C. Michael and et al. *Genetic Algorithms for Dynamic Test-Data Generation*. Technical Report RST-003-97-11 Corporation, 1997.

[19] C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. on Soft. Engin.*, Vol 27(12):1085–1110, Dec. 2001.

[20] L. J. Morell. Theoretical insights into fault-based testing. In *Proc. of Workshop on Software Testing, Verification and Analysis*, pages 45–62. Banff, Canada, 1988.

[21] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[22] A. Offut. The coupling effect: Fact or fiction? In *Proc. of Workshop on Software Testing, Verification and Analysis*, pages 131–140. 1989.

[23] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New-York, EUA, third edition, 1992.

[24] S. Rapps and E. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of International Conference on Software Engineering*. Tokio - Japan, September 1982.

[25] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[26] E. Spinoza and et al. Chameleon: A generic tool for genetic programming. In *Proceedings of the Brazilian Computer Society Conference*. Fortaleza, Brazil, August 2001.

[27] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28(3):157–163, July 1988.

[28] E. Weyuker. Assessing test data adequacy through program inference. *ACM Trans. on Programming Languages and Systems*, Vol. SE-5(4):641–655, 1983.