Selection and Evaluation of Test Data Sets Based on Genetic Programming

Maria Claudia F.P. Emer
mpereira@inf.ufpr.br

Silvia Regina Vergilio
silvia@inf.ufpr.br

Departamento de Informática
Universidade Federal do Paraná - UFPR
CP: 19081, CEP: 81531-970
Curitiba, Brasil

Abstract

A testing criterion is a predicate to be satisfied and generally addresses two important questions related to: 1) the selection of test cases capable of revealing as many faults as possible; and 2) the evaluation of a test set to consider the test ended. Studies show that fault based criteria, such as mutation testing, are very efficacious, but very expensive in terms of the number of test cases. Mutation testing uses mutation operators to generate alternatives for the program P under test. The goal is to derive test cases to producing different behaviours in P and its alternatives. This approach usually does not allow the test of interaction between faults since the alternative differs from P by a simple modification. This work explores the use of Genetic Programming (GP) to derive alternatives for testing P and describes two GP-based test procedures for selection and evaluation of test data. Experimental results show the GP approach applicability and allow comparison with mutation testing.

1 Introduction

Genetic Programming (GP) is a recent research area in the field of Evolutionary Computation. The term was introduced by Koza [15] based on the idea of Genetic Algorithms [13] and on concepts of Darwin's Evolution Theory [4].

GP has the goal of inducing computer programs from examples to solve a given problem. It has been applied to a large number of problems in many areas of Artificial Intelligence and Engineering [1].

In the Software Engineering area, the software test activity has also gained attention during the last decades. This activity is very important to assure the quality of the product being developed and in general consumes a lot of effort. The main goal of testing is to generate test data to reveal as many faults as possible, since is not always possible determine all the faults of a program and to prove its correctness. The complete automation of the test activity has some theoretical limitations, such as: infeasible paths, equivalent programs, coincidental correctness, missing paths, etc.

To guide and systematise the test activity, different testing criteria and tools were proposed to help the tester mainly in two tasks: selection and evaluation of test data sets [21]. The first task is related to the choice of the best points, with high probability of reveal faults. The second one is associated to the following question: How to know whether a program has been enough tested?

The testing criteria are predicates to be satisfied to stop testing, that is, to consider the program tested enough. The existent criteria consider different aspects to select test data and to establish the test requirements. Functional criteria consider the program specification and functionalities. Structural criteria consider the program source code and implementation. They are: control and data-flow based criteria [16, 21]. Fault-based criteria consider the common faults in programming. The best known fault-based criteria is Mutation Analysis [6].

Mutation Analysis establishes for a program P, being tested, a set of mutant programs that differ from P by only a simple modification. The goal is to derive a test case to kill all the generated mutants. To kill a mutant, the test case must to produce different results for P and its mutant. In that case, the mutant is considered dead. A score, given by the number of dead and generated mutants helps the tester to evaluate the used test set. Each transformation in P is described by a mutation operator. An operator is defined according to the programming language and usually transforms statements, variables or constants in the program.

The mutation testing is based on two assumptions: 1) "competent programmer hypothesis" [6]: programmers do their programs very similar to the correct program, according to a specification; and 2) "coupling effect" [6]: assumes that complex faults are detected by analysing simpler faults.

Mothra [5] and Proteum [8] are examples of tools that support mutation testing. Most mutation testing tools and approaches [7, 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). However, Morell [19] points out that these assumptions ignore the global effect of faults or interactions of modifications in the program.

All the testing criteria present advantages and weaknesses; hence, studies addressing their complementary aspects are mandatory [16, 25, 26]. These studies are usually based on three factors [26]: 1) cost: related to the number of necessary test cases; 2) efficacy: the ability to reveal the faults; and 3) strength: related to the difficulty of satisfying a criterion, given that another one has already been satisfied. This last factor is related to the inclusion relation amongst criteria. Structural criteria and mutation testing are theoretically incomparable and only empirical studies can point out the relationship among these criteria, in practice [26].

Some empirical studies show that Mutation Analysis (MA) is more efficacious but more expensive than structural criteria. The cost of MA application is on the number of required executions and on the number of necessary test cases. There are different strategies to apply MA and to reduce its costs. We can mention Randomly Select X% Mutation, Constrained Mutation, etc [17]. They consider different aspects but all of them use the mutation operator approach.

This work presents an approach based on GP to derive alternatives for testing a program P. This approach, named Genetic Programming Based Test (GPBT), like a test criterion, has the goal of helping the tester in the selection and evaluation of test data sets. The idea is to produce the alternatives by using GP instead of mutation operators.

The alternatives do not necessarily differ from P by a simple modification. They can be a combination of more than a mutant and allow the test of interactions between faults.

To apply GPBT is indispensable the use of a tool. Hence, to support GBPT we implemented a tool named GPTesT (**G**enetic **P**rogramming Based **T**esting **T**ool) [10]. Results of an experiment using GPTesT are also presented. They are analysed according to the above mentioned factors: cost, efficacy and strength. They show the applicability of GPBT. The generated test data were also submitted to Proteum for comparison against the mutation operator approach. GPBT allows a reduction in the number of required alternatives and test cases.

The paper is organised as follows. Section 2 shows testing aspects and related works. Section 3 contains an overview of GP: main algorithm, genetic operators and a description of Chameleon, a GP tool, used by GPTesT and to illustrate GPBT. Section 4 introduces the GP based testing approach. It describes, through examples, two GP-based procedures, that are usually supported by a test criterion and tool: selection and evaluation of test cases. Section 5 presents results from the use of GPTesT and Proteum. Finally, Section 6 concludes the paper and discuss future work.

2 Software Test Criteria

Test executes a program with the goal of finding an unrevealed fault. In this sense, input data must be derived for a program P being tested and the selection of these test data is a very important task. Other question related to the testing activity is to known whether a program has being tested enough or how to evaluate a data test set T. These two aspects are discussed by Rapps and Weyuker in [21].

To guide the test activity and answer the above questions, different testing criteria were proposed. Functional criteria use functional specification of a program to derive test cases. 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, 21]. Fault-based criteria derive test cases to show the presence or absence of typical faults in a program, based on common errors in the software development process. The best known fault-based criterion is Mutation Analysis [6].

This work focuses fault-based testing, and the Mutation Analysis criterion will be described in more detail. Mutation Analysis considers two assumptions: 1) the hypothesis of the competent programmer: "Programmers do their programs very similar to the correct program"; and 2) coupling effect: "Tests designed to reveal simple faults can also reveal complex faults". It is also based on a set of mutation operators. The 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. Our goal must be to find a test case set able to kill all non-equivalent mutants. The Mutation Score (MS), 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 [6].

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 [18].

Proteum [8] and Mothra [5] are examples of testing tools based on mutation testing. Proteum has a set of 71 mutation operators and supports test of C programs. Mothra supports testing of Fortran programs.

Theoretical and empirical studies comparing criteria have been conducted based mainly on three factors [26]: 1) cost: it is related to the number of test cases required to satisfy the criterion; 2) efficacy: the ability to reveal faults; 3) strength: it is related to the difficulty of satisfying a criterion, given that another one has been satisfied. This last factor is related to the inclusion relation among criteria.

A criterion C_1 includes a criterion C_2 (Notation: $C_1 \rightarrow C_2$) if, for every program, every test data set which satisfies C_1 also satisfies C_2. If neither C_1 includes C_2 nor C_2 includes C_1, C_1 and C_2 are incomparable [21].

Functional, structural and fault-based criteria are considered complementary, because they can reveal different types of faults. The data-flow based criteria are stronger than control-flow based criteria, that is, they include control-flow based criteria [16, 23, 21]. Structural Criteria and Mutation Analysis are theoretically incomparable and only empirical studies can point out the relationship among these criteria [26].

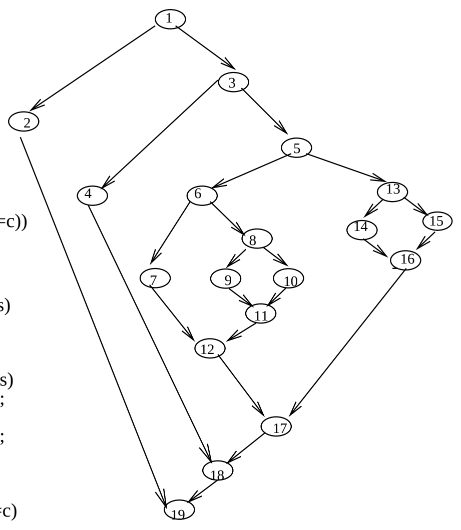
There are a great number of empirical studies [16, 26], comparing different criteria with respect to the mentioned factors [26]. Those studies show that Mutation Analysis is the most efficacious, that is, it has a greater probability of revealing faults, but it is the most expensive in terms of necessary test cases. Other disadvantages are the number of required executions and the existence of equivalent mutants. To reduce the MA costs, different strategies were proposed. We can mention Randomly Select X% Mutation, Constrained Mutation, etc [17]. They consider different aspects but all of them use the mutation operator approach.

Structural criteria, as data-flow based criteria, also have their disadvantages: learning their used concepts is not very easy, and they usually require infeasible elements [23], that is a similar limitation to the equivalence problem. Other limitation inherent to the testing activities and mainly to the structural technique is missing paths. A missing path should exist in the program because it corresponds to a functionality that should be tested, however it is absent. Structural testing derives test cases based on the code and this problem can be not detected. The program triangle [20], presented in Figure 1a, has three integer inputs a,b,c. They correspond to the sides of a triangle. The input must be given in a decreasing order. If this does not happen, the result is -1 (given by the variable *class*). If the numbers do not represent the sides of a triangle, the result is 0. If they really represent a triangle, the result is 1, 2, 3, 4, 5 respectively indicating that the input triangle is equilateral, isosceles, right, obtuse and acute angled. Figure 1b presents the control-flow graph of the program and the paths to be executed. Suppose that the path 1 3 4 18 19 is missing. The test "be or not be a triangle" was not implemented. It is possible to execute all the paths in the graph and not to reveal the fault.

The mentioned limitations hinder the complete automation of the testing activity. Since there is no general procedure to determine infeasible paths or equivalent mutants.

```
     void triangle(a,b,c)
     int a,b,c;
     {  int class;
        float as,bs,cs;
1       if ((a<b) || (b<c))
2          class = −1;
3       else
3       { if (c<=0)
4            class = 0;
5         else
5         { if ((a!=b) && (b!=c))
6              {  as = a*a;
6                 bs = b*b;
6                 cs = c*c;
6                 if (as==bs +as)
7                   class = 3;
8                 else
8                 {
8                    if (as<bs+cs)
9                       class = 4;
10                   else
10                      class = 5;
11                }
12             }
13             else
13             { if ((a=b)&&(b=c)
14                    class = 1;
15                else
15                   class = 2
16             }
17        }
18      }
19      printf("%d", class);
     }
```

a) Code



b) Control Flow Graph

Figure 1: A Missing Path Example

3 Genetic Programming Background

Darwin's Natural Selection Theory [4] shows that, in nature, the individuals that better adapt to the environment that surrounds them have a greater chance to survive. They pass their genetic characteristics to their descendents and consequently, after several generations, this process tends to select naturally individuals, eliminating the ones that do not fit the environment. Genetic Programming (GP) is a field of the Evolutionary Computation, that applies the concepts of evolution to automatically inducing programs. It 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, through recombination of "genes", the program that better solves a given problem. 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. After that, the GP algorithm enters a loop that is executed, ideally, until a desired solution is found. Each run of this loop represents a new generation of computer programs that substitutes the previous one. This process is repeated until a solution is found or until the maximum number of generations is reached.

The tool Chameleon [22] illustrates, in this paper, the use of GP for software testing. Chameleon implements the 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. This operation helps to maintain diversity in the population.

Crossover is the operator that truly performs recombination of computer programs. This operation takes two parents to generate offspring. A random point of crossover is selected on each parent and the sub-trees 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.

Consider the problem of calculating the common minimum multiple of two given numbers (*cmm* problem). Chameleon finds, among other, the solution presented in Figure 2. To execute Chameleon the user needs to give the grammar correspondent to the problem and an initial configuration *I* of parameters. Figure 3 shows the used initial configuration I of parameters, including the grammar adopted to the *cmm* problem. 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 (in this example *cmm.dat*, that contains a set T of training cases. These training cases are used to calculate the fitness value of each individual and describe the expected program output for some input data. The number of runs can be used to end the process. The individual (or program) with better fitness value is selected. The selection can also be random.

```
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

4 Genetic Programming Based Test

Test has the goal of generating input values (test data) from a program. Genetic Programming has the goal of evolving (or generating) programs from input values. We can observe that there is a symmetry between test and GP. Several authors [2, 3, 24] have pointed out this fact by theoretically comparing induction and testing. Based on such symmetry, this section shows how genetic programming can be used to test programs presenting two testing procedures. They show that GP, like a test criterion, helps the tester to answer two test questions: how to select a test case set and how to evaluate a given test set.

```
[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 3: Initial Configuration for Chameleon

Test Data Selection Procedure

Input: a program p to be tested,
an initial configuration I to a GP tool
Output: a set A of alternative programs,
a test data set T

```
$A \leftarrow \emptyset$; $T \leftarrow \emptyset$;
while tester decision is to continue
  $a \leftarrow$ GPtool(I); /* $T \in I$ */
  if $(a \notin A)$
    $A \leftarrow A \cup \{a\}$;
  if $(\exists\ t) \mid p(t) \neq a(t)$
    $T \leftarrow T \cup \{t\}$;
  else
    $E \leftarrow E \cup \{a\}$;
end
```

Initially A and T are empty. In each iteration of the loop, the GP tool generates an alternative program a. The tester must generate a test data t capable of producing different outputs when executes p and a. If such test data t exists it is included on T, otherwise, a is included in the set E of equivalent programs. The test set T is included in the initial configuration I after an iteration. It is necessary to verify whether an alternative a has not already been included in A, because a GP tool can select a program more than once. In that case, the tester can decide whether a new test data is generated. The tester is also responsible for stopping the loop iteration.

Test Data Evaluation Procedure

```
Input: a program $p$ to be tested,
       a finite set $A$ of alternative programs generated by a GP tool,
       a test data set $T$
Output: a coverage score $S_T$

$A' \leftarrow A$; $E \leftarrow \emptyset$;
repeat for each $t \in T$
  repeat for each $a \in A'$
    if $p(t) \neq a(t)$
      $A' \leftarrow A' - \{a\}$;
repeat for each $a \in A'$
  if $(\nexists\ t) \mid p(t) \neq a(t)$
    $E \leftarrow E \cup \{ a\}$;
$S_T = (\#A - \#A')/(\#A - \#E)$;
return $S_T$;
```

The evaluation procedure executes the program p under test and all the alternatives in the set A, generated by the GP tool, for each test case t in the test set T, being evaluated. If p and a produce different results, a is dead and it is excluded. In a second step, the tester identifies the equivalent programs of A. At the end, A' contains only alive no equivalent programs and a score S_T for T is calculated, similarly to the mutation testing.

4.1 Examples

This section illustrates the execution of the procedures using the *factorial* program (Figure 4). Chameleon generated all the alternatives. Table 1 shows the main parameters of Chameleon's initial configuration I and Figure 5 presents the used grammar G. During Chameleon execution, two actions can be adopted: 1) to run Chameleon during a fixed number of generations and to consider all generated population of programs as alternatives; 2) to run Chameleon and to choose only a program from the generated population. The choice can be random or can use the fitness case. The first action was adopted to present the evaluation procedure and, the second, to present the example for the selection procedure.

Table 1: Main Parameters - Initial Configuration I

Population size	1000
Number of runs	100
Tournament size	2
Maximum tree depth for random programs	10
Maximum tree depth during the run	15
Crossover rate	80%
Mutation rate	10%

```
int factorial(int x)
{ int y,i;
  y = 1; i = 1;
  while (i<=x)
  {
    y = y*i;
    i=i+1;
  }
  return(y);}
```

Figure 4: Program *factorial*

```
 ---------------------------------------------------
   <code>   :: = y = <kte>; while (<expb>) {<stmts>}
   <stmts>  ::= <stmt> <stmts>
   <stmts>  ::= <stmt>
   <stmt>   ::= <var> = <var> <opa> <kte>;
   <stmt>   ::= <var> = <var> <opa> <vra>;
   <expb>   ::= <var> <opb> <kte>
   <var>    ::= x | y
   <opb>    ::= < | >
   <opa>    ::= * | +
   <kte>    ::= 1 | 0
---------------------------------------------------
```

Figure 5: Grammar to Factorial Problem

Using GP for Testing Selection

$T \in I$ and initially $T = \emptyset$.

* First iteration of the loop: Chameleon(G,I) is run. The following alternative is chosen.

```
a1(int x)
{ int y;
  y = 0; while(x > 0) { x = y * y; y = y - 1;}
  return (y);
}
```

The tester derives a data test t such $a1(t) \neq p(t)$; t and *a1* are included respectively in T and A. After this, $T = \{0\}$ and $A = \{a1\}$.

* Second iteration of the loop: Chameleon(G,I) is run. The following alternative *a2* is chosen (note that *a2* satisfies T).

```
 a2(int x)
{ int y;
  y = 1; while(x > 0 ) {x = y - 1; y = y * x;}
  return (y);
}
```

The tester derives a data test t such $a2(t) \neq p(t)$; t and *a2* are included respectively in T and A. After this, $T = \{0,1\}$ and $A = \{a1,a2\}$.

* Third iteration of the loop: Chameleon(G,I) is run. The following alternative is chosen.

```
a3(int x)
{ int y;
  y = 1; while(x < 0) { x = x - x; y = x - y; }
  return (y);
}
```

The tester derives a data test t such $a3(t) \neq p(t)$; t and $a3$ are included respectively in T and A. After this, $T = \{0,1,2\}$ and $A = \{a1,a2,a3\}$.

* Fourth iteration of the loop: Chameleon(G,I) is run. The following alternative is chosen.

```
a4 (int x)
{ int y;
  y = 1; while(x > 1) { y = x * 1; x=x-1; }
  return (y);
}
```

The tester derives a data test t such $a4(t) \neq p(t)$; t and $a4$ are included respectively in T and A. After this, $T = \{0,1,2,3\}$ and $A = \{a1,a2,a3,a4\}$

Now the user decides to finish the loop. The solution is given by the sets T and A. A criterion that can be used to take this decision is that whether all the alternatives are equivalent to p, after a given number of generations. The alternative $a5$, presented below, is equivalent to p.

```
a5 (int x)
{ int y;
  y = 1; while(x > 1) { y = x * y; x=x-1; }
  return (y);
}
```

Using GP for Evaluating a Test Data Set

The GP-based approach can be used as a criterion to assess the quality of a test data set. For example, consider two test sets T_1={0,1,2} and T_2={0,1,2,3}, and the question: Which set is better?

We can answer the question using the set of alternatives $A = \{a1,\ a2,\ a3,\ a4,\ a5\}$, generated by GP/Chameleon, and choosing the set with the greatest score S. Executing the second procedure for both test sets, and being E determined by the tester, the results are:

T_1: Initially $A' = A$ and $E = \emptyset$.
At the end, $E = \{a5\}$ and $S_{T_1} = 0.75$.

T_2: Initially $A' = A$ and $E = \emptyset$.
At the end, $E = \{a5\}$ and $S_{T_2} = 1$.

Analysing the results, the test set T_2 is considered adequate and "better" than T_1 according to A. S_{T_2} is greater than S_{T_1}.

5 Evaluating GPBT

As observed in Section 3, the complete automation of a testing criterion is impossible due to many testing limitations. Considering these limitations, a tool, named GPTesT (**G**enetic **P**rogramming-based **Tes**ting **T**ool) [10] was implemented to support the GPBT approach, described in last section. GPTesT uses Chameleon to derive the alternatives.

This initial version of GPTesT allows the unit testing of programs in C language. GPTesT, as well Chameleon, is 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). More details about GPTesT implementation are in [10].

This section describes an experiment accomplished with GPTesT. The goal is to evaluate GPBT and to allow comparison against mutation testing using Proteum operators. This experiment was first described in [9, 11]; here with the perspective of strength analysis, we add a new step (Step 6).

5.1 Description of the Experiment

We used four programs described in Table 2. The same steps were followed using Proteum and GPTesT; they are next:

Table 2: Programs Used for GPBT Evaluation

Programs	Description
cmm	prints the common minimum multiple of two given numbers
fat	prints the factorial of a given number
max	prints the largest of its three inputs
cmd	prints the common maximum divisor of two given numbers

1. generation of the alternatives: we use all the operators available in Proteum. GPesT uses the alternatives generated by Chameleon with the basic configuration presented on Table 3. This configuration was used for all programs, however, the grammar and the initial training cases are dependent on the specific problems.

Table 3: Main Chameleon Parameters

Population size	500
Number of runs	100
Tournament size	7
Maximum tree depth for random programs	30
Maximum tree depth during the run	60
Crossover rate	90%
Mutation rate	0%

2. generation of test cases and submission to Proteum and GPTesT: two different testers generated the sets. So, the test cases submitted to the tools are not the same.

3. execution of the program under test and its alternatives: as a result an initial score were obtained.

4. identification of equivalent programs: the equivalence was manually determined.

5. generation and submission of additional test cases: the final scores were obtained. Table 4 presents the results obtained for both tools. For example, for program *fat*, Chameleon generated 814 alternatives, but GPTesT discarded 412. There are not anomalous alternatives in this set, as well, equivalent. The adequate set to kill all the alternatives has 5 test cases, that is, all of them are effective and really contribute to increase the score (in this test set is not included the training set of configuration I). Proteum generated 272 mutants, 0 anomalous, but we identified 29 equivalent mutants; it was also necessary 5 test cases.

6. strength analysis: the GP adequate test sets were submitted to Proteum to analyse the difficult of satisfying the Mutation Analysis criterion. Proteum scores obtained using GP sets are in Table 5. The Proteum adequate test sets were submitted to GPTesT and the results are also presented. This table shows the number of test cases from the adequate test sets that were effective for both tools.

Table 4: Main Results from GPTesT and Proteum

Program	Tool	# Alternatives Generated	# Active Alternatives	# Anomalous	# Equivalent	# Effective Test Cases
fat	GPTesT	814	402	0	0	5
	Proteum	272	272	0	29	5
max	GPTesT	170	115	0	7	6
	Proteum	527	527	0	74	28
cmm	GPTesT	525	44	0	0	4
	Proteum	551	551	0	40	16
cmd	GPTesT	464	152	0	0	7
	Proteum	397	397	0	33	14
Total	GPTesT	1973	713	0	7	22
	Proteum	1747	1747	0	176	63

Table 5: Strength Analysis

Program	GP in Proteum		Proteum in GPTesT	
	Score	#Effec. TC	Score	#Effec. TC
fat	0.94	5/5	1	3/5
max	0.72	5/6	1	6/28
cmm	0.90	4/4	1	2/16
cmd	0.90	7/7	1	7/14

5.2 Analysis of the Results

- Number of Alternatives: the total number of alternatives generated by GPTesT is greater than Proteum. However, if we consider only active alternatives, that is, no anomalous and different alternatives, Proteum generated 2 times more alternatives than GPTesT.

- Generation of Alternatives: to generate alternatives by GPTesT, the tester has to give the grammar associated to the problem and the training test cases for Chameleon. The quality of the generated alternatives is dependent on this tester task. We observed that this task is very important and consumes more effort that to choose the mutations operators in Proteum. Chameleon is grammar-oriented, hence, if the grammar does not appropriately describe the problem, the competent programmer hypothesis will be not considered and maybe all the generated alternative will be dead with only one test case. This fact did not happen during our studies, the alternatives generated by Chameleon are very close to the solution.

- Number of Test Cases: for any program, GPTesT required a lower number of test cases than Proteum, even for *fat* program that presents a greater number of GPTesT alternatives. The total number of effective test cases required by Proteum is 2.5 times greater.

- Anomalous Alternatives: both tools has a mechanism to discard anomalous from the set of generated mutants.

- Equivalent Mutants: GPTesT generated only 1% of the equivalent alternatives. Proteum generated 11%. The determination of equivalent mutants represents a lot of work and effort spent with Proteum.

- Runtime: GPTesT and Proteum execute in different environments and operational systems. Hence, we do not compare the runtime. Proteum generates a greater number of alternatives, so the bottleneck is on the time for executing the mutants. The GPTest bottleneck is associated to Chameleon execution, because the number of alternatives is lower.

- Efficacy: during the experiment we found an error. Many programs do not consider negative inputs. This fact was pointed out by both tools. However, we intend to conduct other experiments with incorrect versions to evaluate this factor.

- Strength Analysis: all the GP strengths in Proteum except for program *max* are high. A possible reason for this is that the program *max* presents the greatest difference in the number of alternatives and necessary test cases. In that case, Proteum generates 4.5 more mutants than GPTesT, and requires 4.6 more test cases. Other point to be observed is that only a test case in the GP adequate test sets were not effective in Proteum.

  The Proteum adequate test sets, in other hand, always killed all the non equivalent GPTesT alternatives, and with a lower total number of test cases (18) than the total number used by the tester (22). This fact points out the validity of the coupling effect hypothesis. The simple faults described by the Proteum operators are capable of describe the composed faults described by the alternatives genetically derived.

6 Conclusions

This work explored the use of Genetic Programming (GP) in the software test activity. An approach for software test based on GP is presented. The approach is similar to Mutation Analysis criterion but the concepts of evolution are used instead mutant operators. The mutation operator approach generates mutants that differ from P by only a simple modification. GPBT tests interactions between faults and permits the combination of two or more mutants as only one alternative. The mutation operators available, for example, in Proteum and Mothra tools can not derive all the alternatives used in the example of Section 4.1.

This paper described two testing procedures that show how the GPBT can be used to help the tester in the task of selection and/or evaluation of a test data set. They are a basic requirement supported by most test criteria and tools.

The test data selection procedure has the advantage of considering the test sets given by the user during the alternative generation process. This reduces the number of anomalous and alive alternatives, and consequently the effort spent in the test. The initial population may be very different from the correct program, and Chameleon executions depend on the grammar and the training test cases given by the user. The user decides how much the hypothesis of the competent programmer should influence the alternative generation. However, this fact is not a disadvantage. The effects of a missing path decrease because the code is not used to derive the alternatives. This is also 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 criteria, all the required elements must be generated again.

GPBT is independent of the language or paradigm used for the program being tested. GPTesT uses Chameleon and C language, however, can easily extend to support the use of other tools and paradigms found in the literature, for instance, that evolve LISP programs.

With respect to the test data evaluation procedure, there is an important point to be analysed. The score is strongly based on the set of alternatives A, which is an input to the procedure. Different scores are generated if different sets of alternatives are considered. Obviously the mutation score also depends on the used mutation operator, but the dependence is different here. Two executions of Chameleon do not guarantee the generation of the same alternative set, even using the same initial configuration. This must be considered when evaluating test data sets and should be evaluated in future experiments. It is not a disadvantage, because it can be used as another factor to compare test data sets.

The results from the experiment show the GPBT applicability. We observed a reduction in the number of generated alternatives and test cases. Both adequate test sets revealed the problem with negative numbers. This points out that the lower GPBT cost did not influence the efficacy maybe due to an alternative be a combination of one or more mutants. However, we intend to evaluate the factor efficacy in future experiments. Other result is related to the percentage of equivalent programs: 1% for GPTesT against 11% for Proteum. Determining equivalent mutants is a tedious task that consumes a lot of effort. In out experiment GPBT decreased this effort.

The GPBT score in Proteum is around 0.9 for most programs. We used all the operator in Proteum. Most recent studies determined a set of essential operators in Proteum to establish a MA strategy with lower costs. Other experiments with this and other strategies, mentioned in Section 2 should be conducted with the goal of evaluating strength and efficacy.

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.

Chameleon evolves simple programs and this is a current GPTesT limitation. In spite of this, GPBT and GPTesT are as promising as the GP field. New advances in GP will contribute to improve GPTesT and to increase the GPBT applicability. For instance, Chameleon is being now extended to support evolution of sub-programs. This will allow GPTesT extension to support integration test.

References

[1] *Proceedings of Genetic and Evolutionary Computation Conference.* Morgan Kaufmann Publishers, 2001/2002.

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

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

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

[5] R.A. De Millo, D.C. Gwind, and K.N. 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.

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

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

[8] M. E. Delamaro and J.C. 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.

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

[10] M.C.F.P. Emer and S.R. Vergilio. Gptest: A testing tool based on genetic programming. In *Proceedings of Genetic and Evolutionary Conference - GECCO*, pages 1343–1350. Morgan Kaufammn Publishers, New York, July 2002.

[11] M.C.F.P. Emer and S.R. Vergilio. Software test using genetic programming. In *Annals of Software Engineering- Special Volume.* 2002. (submitted).

[12] F.G. 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.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1992.

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

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

[16] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, July 1991. (in Portuguese).

[17] A.P Mathur and W.E. Wong. Evaluation of the cost of alternate mutation strategies. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 320–375. Rio de Janeiro-Brazil, October 1993.

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

[19] 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.

[20] C.V. Ramamoorthy, F. H. Siu-Bun, and W.T. Chen. On automated generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):293–300, December 1976.

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

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

[23] S.R. Vergilio, J.C. Maldonado, and M. Jino. Infeasible paths within the context of data flow based criteria. In *VI International Conference on Software Quality*, pages 310–321. Ottawa-Canada, October 1996.

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

[25] E.J. Weyuker. An empirical study of the complexity of data flow testing. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 188–195. Computer Science Press, Banff - Canada, July 1988.

[26] W.E. Wong, A.P. Mathur, and J.C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *Software Quality and Productivity - Theory, Practice, Education and Training.* Hong Kong, December 1994.