# Applying Mutation Testing in Prolog Programs

Juliano R. Toaldo[1] and Silvia R. Vergilio[1]

Federal University of Parana (UFPR), CP: 19081,
CEP: 81531-970, Curitiba - Brazil
jtoaldo@yahoo.com.br, silvia@inf.ufpr.br

**Abstract.** Several testing criteria and tools have been proposed lately, with the goal of selecting and evaluating test data sets. However, most works focus only procedural and object-oriented programs and little has been said about logic programming languages, such as Prolog. Some works address the test of Prolog programs however, do not introduce a testing criterion and not offer coverage testing metrics. This work investigates the application of the Mutation Analysis criterion for testing Prolog programs. In experiments of literature, this criterion has been considered one of the most efficacious. A set of mutation operators for this language is proposed, based on common mistakes made by the programmers using this paradigm. A tool, named MutProlog, is described. This tool supports the proposed operators and eases the development of Prolog programs. Results from an experiment, using MutProlog, show the applicability of the proposed operators and allow comparison with structural criteria.

*Keywords*: testing criteria, logic programming, mutation testing.

## 1 Introduction

In the software development process, software testing is one of the most important activities for software quality assurance. However, the testing activity is very expensive; the testing teams should be properly trained and in some cases, adequate tools are not available. Because of this, some works introduced testing techniques and criteria, with the goal of revealing a great number of faults with minimal effort and costs.

Since to execute the program for all inputs of its domain, that is, to perform an exhaustive testing is not always possible, the testing criteria were proposed to help the tester in the task of selecting test data and/or of evaluating a given test set T. A testing criterion is a predicate to be satisfied to consider the testing activity ended [19]. It offers a metric, based on the coverage of certain elements, named required elements. This metric is used to consider whether a program has been tested enough. For example, functional criteria consider functional aspects of the program. Structural criteria, such as control and data-flow based criteria [19], consider internal aspects of the program or the specification to generate the

© *A. Gelbukh, C. Yáñez Márquez, O. Camacho Nieto (Eds.)*
*Advances in Artificial Intelligence and Computer Science*
*Research on Computing Science 14, 2005, pp. 81-92*

test data. Fault-based criteria derive test data to show the presence or absence of typical faults in a program, based on common errors in the software development.

The different criteria are considered complementary because they can reveal different kind of faults. However, some empirical studies show that the fault-based criteria are the most efficacious to reveal faults [16,20].

In spite of this great number of testing criteria and supporting tools, most of them focus conventional procedural and object-oriented paradigms. Few works are dedicated to logic programming languages, such as Prolog.

The majority of the works about test of Prolog programs, generates test data considering only functional aspects [4,10,13,14]. The work of Bergadano et al [2] uses ILP (Inductive Logic Programming) and the works described in [3,17,18] are related to debugging and detection of anomalies. These works do not extend the testing criteria and not allow the use of coverage metrics. To overcome this limitation, Luo et al [15] propose the extension of structural criteria for testing Prolog programs. They propose a control-flow graph for Prolog and two criteria based on this graph: all-branches, and all pairs branch-to-branch.

In a complementary way, our work investigates the use of fault-based testing of Prolog programs, particularly the use of mutation testing [7]. Mutation Analysis (MA) is based on two assumptions [7]: 1) "competent programmer hypothesis", e.g. programmers do their programs very similar to the correct program, according to a specification. 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 (mutants) for P that can include at least one correct program; and 2) coupling effect: complex faults are usually revealed by revealing simpler ones.

Mutation testing [7] consists basically of generating mutant programs for the program P being tested. 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 data set T. If a mutant M presents different results from P it is said to be dead. In the last case, either there is no test data in T that is capable to distinguish M from P, or M and P are equivalent. Our goal must be to find a test data set able to kill all non-equivalent mutants. The mutation score allows the adequacy evaluation of T.

The existence of equivalent mutants is a limitation to determine the mutation score, because there is no algorithm to determine whether two programs compute the same functions. Similar limitation is found when applying structural criteria. Some paths required by these criteria are infeasible and they can not be automatically determined. The identification of infeasible paths and equivalent mutants are un-decidable questions [1,5,12].

In spite of the above limitations, we find in the literature tools that support structural and fault-based criteria To allow the application of mutation testing, a tool is fundamental. Proteum [9] and Mothra [6] are examples of tools that implement Mutation Analysis, respectively for C and FORTRAN languages.

To permit the application of mutation testing in Prolog programs it is necessary the existence of mutation operators for this language and a tool to make

possible the automatic execution of the mutants and the evaluation of the test data sets. With this goal in mind, this paper introduces a set of mutation operators for Prolog and describes a tool, named MutProlog that supports the proposed operators and the application of mutation testing in this context.

The work is organized as follows. Section 2 describes related works that address the test of Prolog programs. Section 3 describes a fault model for Prolog programs and, based on this model, introduces the set of mutation operators. Section 4 shows functional aspects of MutProlog. Section 5 describes procedures of use for MutProlog. Section 6 presents the main results of an experiment conducted for evaluation of the introduced operators. Section 7 concludes the paper.

## 2 Related Works

In the last decades, many works addressed the testing activity. Several testing criteria and tools were proposed to easy the test. However, most of them focus only procedural and object-oriented languages. Few works explore these criteria in the logic or functional programming languages.

The works described in [4,10,13,14] generate test cases based on functional aspects of the specification. Other works are related to debugging and detection of anomalies [3,17,18]. The work of Bergadano et al [2] uses ILP (Inductive Logic Programming) for generating the test data without offers a coverage metric. Emer and Vergilio proposes the use of Genetic Programming [11] for generating mutant programs. They comment that the approach can be used for any paradigm, however it is necessary a tool based on genetic programming for evolving complete programs. The authors only explore the test of C programs.

Luo et al [15] explore the use of structural criteria for testing Prolog programs. They propose a control-flow graph for Prolog and two criteria based on the graph: all-branches, and all pairs branch-to-branch. To satisfy these criteria, it is necessary to execute the program with test data that exercise paths in the graph. The paths must to include all the branches in the graph and all pairs of branches. In some cases, the required elements (branches or pair of branches) can be infeasible, if all the paths that exercise them are also infeasible. The tool TGT (Test Data Generation Tool) was implemented to validate the structural criteria.

The work of Luo et al is the most similar to ours. In next section, we also explore the use of criteria for Prolog programs. However, we propose the use of Mutation Analysis, that is a fault-based criterion. The main motivation to do this is that results from experiments reported in the literature [16,20] show that this criterion is the most efficacious and we did not find any work that explores these criteria in the mentioned context.

## 3 Mutation Operators for Prolog

This section introduces a set of mutation operators for Prolog based on our experience as programmers and on the characteristics of Prolog programs [15].

- the data structures are recursive lists, recursion is very used in Prolog.
- the unification of sub-goals in Prolog can proceed on two directions; the existence of backtracking is a very important characteristic.
- there are not pre-defined types for the variables, they match with different kind of variables, there is the anonymous variable.
- there are no routines, a set of clauses is used, and the concept of unit testing needs to be redefined.

These characteristics are related to the main mistakes made by the programmers that most frequently do not have the explicit control. Considering these aspects, a classification for the main non syntactical faults of the programs established and a mutation operator for each class is introduced. They were classified in four groups. Table 1 presents a description and examples showing the transformation of the program caused by each operator in each group.

1. clause mutation: each Prolog rule, finalized with a "." is considered a clause. This group includes changes in conjunction and disjunction, operations with *cut* ("!"), changes in predicates, etc. The changes in predicates happens only in adjacent predicates, because we are considering the hypothesis of the competent programmer [7]. However, changes in operations with *cut* happens belong all the programs; errors using *cut* are much more frequent.
2. operator mutations: differently of conventional languages there are no many operators in Prolog. We include the arithmetic and relational ones. The idea is to change them by similar operators.
3. variable mutation: two types of variable are considered: anonymous or not. A variable is changed by other one in the same clause, independently of its type.
4. constant mutation: constants are changed by other constants or variables any type in the same clause.

The introduced operators are capable of revealing the faults that they describe. However, based on the coupling effect assumption [7], complex faults are combination of simpler faults and can be detected when simple faults are revealed. We illustrate this fact with the program *merge* [2] (Figure 1a) and its incorrect version with only four clauses (Figure 1b). The incorrect program does not eliminate duplicates. A test set satisfying all-branches criterion may not reveal the fault. However, a test data selected to kill the mutants generated by the operator "Relational Operator" necessarily reveals the fault.

**Table 1.** Mutation Operators

| Operator | Description | Original Program | Example of Mutant |
|---|---|---|---|
| | | **Group 1: Clause Mutations** | |
| Delete Predicate | remove predicate P in clause C | writel([]).<br>writel([H\|T]) :- write(H),nl,<br>writel(T). | writel([]).<br>writel([H\|T]) :- nl,<br>writel(T). |
| Swap Predicate | change the order in adjacent predicates | likes(ana,X) :- toy(X),<br>plays(ana,X). | likes(ana,X) :- plays(ana,X),<br>toy(X). |
| Conjunction by Disjunction Replacement | change conjunction of predicates by disjunction | subset(S,[H\|T]) :- subset(R,T),<br>(S=R , S=[H\|R]).<br>subset([],[]). | subset(S,[H\|T]) :- subset(R,T),<br>(S=R ; S=[H\|R]).<br>subset([],[]). |
| Disjunction by Conjunction Replacement | change disjunction of predicates by conjunction | least_num(X,[H\|T]) :-<br>least_num(Y,T),<br>(H=<Y,X=H,H>Y,X=Y). | least_num(X,[H\|T]) :-<br>least_num(Y,T),<br>(H=<Y,X=H;H>Y,X=Y). |
| Insert Cut | insert cut between predicates | bubble_sort(L,L1) :-<br>swap(L,L2,0),<br>bubble_sort(L2,L1).<br>bubble sort(L,L). | bubble_sort(L,L1) :-<br>swap(L,L2,0),!,<br>bubble_sort(L2,L1).<br>bubble sort(L,L). |
| Remove Cut | remove cut operator | not(G) :- G,!,fail.<br>not(G). | not(G) :- G,fail.<br>not(G). |
| Permute Cut | change the place of a cut | insert(X,[H\|T],[H\|T1] :- !,X>H,<br>insert(X,T,T1). | insert(X,[H\|T],[H\|T1] :- X>H,!,<br>insert(X,T,T1). |
| | | **Group 2: Operator Mutation** | |
| Arithmetic Operator Mutation | change an arithmetic operator by other arithmetic operator | length([],0).<br>length([_\|T],N) :-<br>length(T,M),N is M+1. | length([],0).<br>length([_\|T],N) :-<br>length(T,M),N is M*1. |
| Relational Operator Mutation | change a relational operator by other relational operator | fault(X) :-<br>non(respond(X,Y)),X\ ==Y. | fault(X) :-<br>non(respond(X,Y)),X>Y. |
| | | **Group 3: Variable Mutation** | |
| Variable by Variable | change a variable in clause C by other one in C | member(X,[T\|_]).<br>member(X,[_\|T]) :- member(X,T). | member(X,[X\|_]).<br>member(X,[_\|T]) :- member(X,T). |
| Variable by Anonymous Variable | change a variable by an anonymous variable | length([],0).<br>length([H\|T],N) :-<br>length(T,M),N is M+1. | length([],0).<br>length([_\|T],N) :-<br>length(T,M),N is M+1. |
| Anonymous Variable by Variable | change anonymous variable in clause C by other variable in C | member(X,[X\|_]).<br>member(X,[_\|T]) :- member(X,T). | member(X,[X\|T]).<br>member(X,[_\|T]) :- member(X,T). |
| | | **Group 4:Constant Mutation** | |
| Constant by Constant | change a constant in clause C by other one | length([],0).<br>length([_\|T],N) :-<br>length(T,M),N is M+1. | length([],1).<br>length([_\|T],N) :-<br>length(T,M),N is M+1. |
| Constant by Anonymous Variable | change a constant in clause C by an anonymous variable | likes(ana,jose).<br>likes(jose,ana). | likes(ana,jose).<br>likes(jose,_). |
| Anonymous Variable by Constant | change anonymous variable in clause C by a constant in C | likes(ana,apple).<br>likes(paulo,_). | likes(ana,apple).<br>likes(paulo,apple). |
| Constant by Variable | change a constant in clause C by a variable in C | append([],L,L).<br>append([H\|T1],L2,[H\|T]) :-<br>append(T1,L2,T). | append(L2,L,L).<br>append([H\|T1],L2,[H\|T]) :-<br>append(T1,L2,T). |
| Variable by Constant | change a variable in clause C by a constant in C | nth(N,X,[X\|T]).<br>nth(N,X,[Y\|T]) :-<br>nth(M,X,T),N is M+1. | nth(1,X,[X\|T]).<br>nth(N,X,[Y\|T]) :-<br>nth(M,X,T),N is M+1. |

```
merge(A,[],A).
merge([],B,B).
merge([A|],[B|Rb],[A|M]) :- A < B,
          merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[A|M]) :- A = B,
          merge(Ra,Rb,M).
merge([A|Ra],[B|Rb],[B|M]) :- A > B,
          merge([A|Ra],Rb,M).
```

```
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :- A <=
          merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :- A > B,
          merge([A|Ra],Rb,M).
```

**Fig. 1.** Program *merge* a) Program original. b) Incorrect Version.

# 4 MutProlog

The complete automation of a testing criterion is impossible due to many testing limitations. Considering these limitations, a tool, named MutProlog was developed to support the introduced operators and to allow the fault-based testing. MutProlog was implemented in C language and operational system Linux. It has four modules, illustrated in Figure 2 and described below.

*Generate mutants*: receives the source code in SWI Prolog and the configuration file. It produces a directory(*Mutants*), that contains files and descriptions for the transformations to be applied. All the clauses in the source code are considered to generate the mutants individually. The configuration file contains the percentage to be applied for each operator. For example, for program *merge*, a configuration file from Figure 3 indicates a percentage of 50 to the operator "Insert cut". This means, if for a clause, 4 mutants can be generated applying this operator, only 50% (2) are generated.

*Generate scripts*: generates a script for automatic execution of the mutants. It is executed after the generation of the mutants. This module uses the parameters "input_variable" and "output_variable" given in the configuration file. For example, *merge* has two input variables and only one output.

*Execute mutants*: this module is used in two cases. In the first one, the user adds test data to the existent ones, and the source code is executed, its input and outputs are saved in a directory. In the second case, the mutants are executed and their outputs are compared with the original outputs. This module can be executed many times, but only the alive and enabled mutants are executed. To each mutant is associated to a status that can be: dead, alive, anomalous or equivalent. Mutants that produce an execution error, such as division by zero, are identified as anomalous. Equivalent mutants produce the same original output for all inputs; they are identified by the tester, because determining equivalence between programs is an un-decidable question [1,5]. The tester can also enable or not a test data.

*Evaluate mutants*: the coverage is calculated using the formula below:

$$MS(P,T) = \frac{M_d(P,T)}{M(P) - M_e(P)}, where:$$

- $P$: program under test;
- $T$: set of given test data;
- $MS(P,T)$: mutation score;
- $M_d(P,T)$: number of dead mutants;
- $M(P)$: number of no anomalous generated mutants;
- $M_e(P)$: number of equivalent mutants.

According to the obtained coverage the tester decides to stop testing or to add more test data to reach the desired coverage.

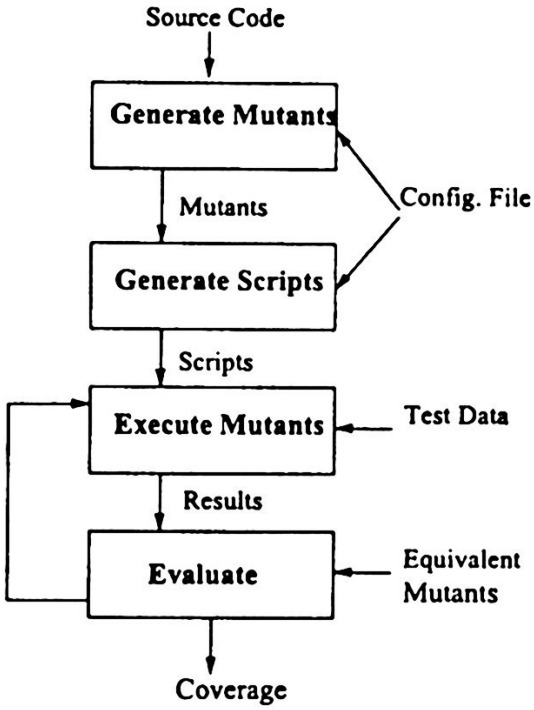


**Fig. 2.** Main Modules of MutProlog

```
input_variable=X
input_variable=Y
output_variable=Z
ClauseMutationPredicateDeletion=100
ClauseMutationSwapPredicates=100
ClauseMutationConjunctionByDisjunctionReplacement=100
ClauseMutationDisjunctionByConjunctionReplacement=100
ClauseMutationInsertCut=50
ClauseMutationRemoveCut=100
ClauseMutationPermuteCut=100
OperatorMutationArithmetic=100
OperatorMutationRelational=100
VariableMutationVariableByVariable=100
VariableMutationVariableByVariableAnonymous=100
VariableMutationVariableAnonymousByVariable=100
ConstantMutationConstantByConstantReplacement=100
ConstantMutationConstantByVariableAnonymous=100
ConstantMutationVariableAnonymousByConstant=100
ConstantMutationConstantByVariable=100
ConstantMutationVariableByConstant=100
```

**Fig. 3.** Configuration File for *merge*

# 5 An Example of Use

As mentioned in Section 2, a testing criterion requires a set of elements to be exercised by the test data. It can be used for selection or evaluation of test data sets. In the case of MutProlog, the required elements are mutants programs that should be dead by the test data.

## 5.1 Selection of Test Data

Suppose that we have to test a program P, but we do not have any test case set. We can use Mutation Analysis and MutProlog as a guideline for the selection test data. We should conduct the steps following steps:

1. Create a configuration file for the program.
2. Generate the mutants, using MutProlog.
3. For each generated mutant, generate a test data t to produce different outputs executing P and the mutant. If such case does not exist, set the mutant as equivalent.
4. Generate scripts using MutProlog and execute the mutants and P with t. The test data is saved. If the output produced by P is not correct. The fault needs to be removed and a regression testing is necessary. If the output produced by a mutant is correct, it is very easy to correct the original program, since P and the mutant differs only by a syntactical change.
5. If the outputs are really different, then the mutant is really dead. MutProlog calculates the mutation score (or coverage).
6. If the desired score was obtained, stop testing. Otherwise, choose another mutant and repeat Steps 3, 4 and 5.

At the end of the process we have tested P and now we have a set T of test data generated using the Mutation Analysis criterion and a level of reliability, given by the obtained coverage of T. Observe that if we have an initial set Step 3 is not necessary, we can execute the mutants, check the outputs and calculate the mutation score for all the test data in T. After this, we decide continue or not, using MutProlog to improve T. This allows the combination of other testing techniques with fault-based testing. The initial set of test data could be generated using, for example, functional or structural techniques.

## 5.2 Evaluation of Test Data Sets

A testing criterion is also used to assess the quality of a test dataset. For example, consider two test sets: $T_1$ and $T_2$, and the question: Which set is better? We answer the question using MutProlog and the following steps:

1. Create a configuration file for the program.
2. Generate the mutants, using MutProlog.
3. Generate the scripts and execute the mutants and the original program using $T_1$ and $T_2$.

4. Calculate the scores and choose the test set with the greatest score.

This procedure can also be used to evaluate a particular test set T, to know how good it is by considering the Mutation Analysis criterion.

## 6 Experiments

The experiment used four programs (*elem_rep*, *num_ap*, *ord_sel* e *merge*). The goal of the experiment is a preliminary evaluation of the proposed operators. The steps below were followed for every program:

1. generation of the configuration file and of the mutants. All the operators were applied 100%.
2. generation of test data, execution of the mutants e evaluation of the results.
3. generation of additional test data to kill the alive mutants and determination of the equivalent mutants.
4. execution of the mutants with the new test data.
5. repetition of the last two steps until all non-equivalent mutants are dead. At the end of this step, we obtained MutProlog-adequate test sets ($T_{mp}$), composed only by test data that really killed a mutant. Table 2 shows the obtained results. For program *merge* 186 mutants were generated, 171 died with 4 test data and 15 are equivalents. $T_{mp}$ for *merge* has 4 elements.
6. generation of the control-flow graphs for all programs and of the required elements for the all-branches criterion.
7. identification of infeasible structural elements. Table 3 shows the required and infeasible branches found.
8. evaluation of $T_{mp}$. Table 4 shows the results of this step. The $T_{mp}$ sets always get a 100% coverage of all branches. That is, the $T_{mp}$ sets are all-branches adequate. We can observe in that table, that some test data of $T_{mp}$ do not contribute to cover any branch. Only 6 test data are necessary. Considering only these necessary test data, we obtained $T_b$ sets, all-branches adequate test sets, which are sub-sets of $T_{mp}$.
9. submission of $T_b$ sets in MutProlog. The scores are in the last colunms of Table 4.

The numbers in Table 2 are very small when compared with the ones obtained in experiments with traditional programs. The number of generated mutants and necessary test data are very small. This happens because Prolog programs are smaller than C programs.

The percentage of equivalent mutants found, around 7%, is also low, showing that the operators set generated a small number of equivalent mutants. This is very important, because the manual determination of equivalent mutants spends a lot of effort and time in the testing activity. We observe that there are no infeasible branches.

To compare mutation testing and structural testing we used three factors, usually used in works from the literature: cost, given by the number of test data; strength, related to the difficulty of satisfying a criterion, given that another one was satisfied; and efficacy, related to the number of revealed faults.

**Table 2.** Generated and Equivalent Mutants and, Required Test data

| Program | Generated Mutants | Equivalent Mutants | Test data ($T_{mp}$) |
|---|---|---|---|
| *elem_rep* | 314 | 21 (6,68%) | 3 |
| *num_ap* | 208 | 18 (8,65%) | 3 |
| *ord_sel* | 187 | 9 (4,81%) | 3 |
| *merge* | 186 | 15 (8,06%) | 4 |
| Total | 895 | 63 (7,03%) | 13 |

**Table 3.** Required and Infeasible Branches

| Program | Required | Infeasible |
|---|---|---|
| *elem_rep* | 17 | 0 |
| *num_ap* | 14 | 0 |
| *ord_sel* | 10 | 0 |
| *merge* | 26 | 0 |
| Total | 67 | 0 |

- *Cost*: the cost of mutation testing is 2 times greater than the cost of structural criterion. In traditional programs this cost can be until 3 times more, because of this, the obtained cost was smaller than we expected.
- *Strength*: The MutProlog adequate sets covered 100% of the required branche However, 10% of the non-equivalent mutants were not killed by the branches adequate sets. This can mean that the all-branches adequate may be not include test data capable to reveal the faults described by operators that generated those not dead mutants.
- *Efficacy*: With the goal of evaluating the efficacy of both criteria, we created wrong versions for the programs, introducing one or more faults in each program. Table 5 shows the total number of faults introduced in a random way in each program. The wrong versions were executed by the $T_b$ and $T$ sets. The $T_{mp}$ sets found one error (5%) more than the $T_b$ sets.

## 7 Conclusions

This work addressed the application of Mutation Analysis for testing Prolo programs and described a supporting tool.

We introduced a set of mutation operators, based on Prolog characteristic and on typical faults found in the programs. As illustrated in Section 4, the operators allow the discovering of the faults described, as well as their combinatior

A tool that implements the operators and supports the practical applicatio of the criterion was described. MutProlog generates and executes the mutant automatically, and calculate the coverage obtained for a given test data set. use of MutProlog can ease the development of Prolog programs and to reduc the cost of the testing activity.

The results of the experiment accomplished are very promising and sho the applicability of the mutant operators proposed. The percentage of generate

**Table 4.** Strenght Results

| Program | MutProlog X All-Branches | | All-Branches X MutProlog | |
|---|---|---|---|---|
| | Test data | Control Flow Coverage | Test Data | MutProlog Coverage |
| *elem_rep* | 1 | 100% | 1 | 79,8% |
| *num_ap* | 1 | 100% | 1 | 93,75% |
| *ord_sel* | 1 | 100% | 1 | 92,5% |
| *merge* | 3 | 100% | 3 | 97,07% |
| Total | 6 | 100% | 6 | 90,25% |

**Table 5.** Number of Faults Revealed by the Test Sets

| Program | Faults | Revealed faults $T_b$ | Revealed faults $T_{mp}$ |
|---|---|---|---|
| *elem_rep* | 5 | 4 (80%) | 5 (100%) |
| *num_ap* | 5 | 5 (100%) | 5 (100%) |
| *ord_sel* | 5 | 4 (100%) | 4 (80%) |
| *merge* | 5 | 5 (60%) | 5 (100%) |
| Total | 20 | 18 (90%) | 19 (95%) |

and equivalent mutants is lower than for traditional programs, because Prolog programs are usually smaller.

When compared with structural testing, the results are similar to traditional programs. Mutation testing requires a greater number of test data than the all-branches criterion. The strength results show that to satisfy the mutant criterion is harder than the structural criterion. We have a greater probability of satisfying all-branches criterion if the mutant criterion was satisfied.

The results also indicate a greater efficacy but new studies should be conducted. This is only a preliminary work. The set of operators herein introduced should be better investigated. New operators should be proposed, mainly to help the inter-clause testing. These operators can be proposed based on the concept of interface mutation applied to integration testing [8].

In a second step, we intend to accomplish an experiment to investigate essential operators for Prolog. Sets of operators could be established according to some aspects and kind of application related to the program being tested. This has been successfully done in conventional programs for decreasing costs.

Some improvements in MutProlog are necessary, such as, the development of a graphical interface and mechanisms to help the tester in the identification of equivalent mutants and in the automatic generation of test data.

## References

1. D. Baldwin and F. Sayward. *Heuristics for Determining Equivalence of Program Mutations.* CT, Res.Rep. 276, Department of Computer Science - Yale University, New Haven, 1979.
2. F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1996.

3. P. Boeck and B. Charlier. Static type analysis of prolog procedures for ensuring correctness. In *International Workshop PLILP*, pages 223–237. Spring-Verlag, Lectures Notes in Computer Science, 456, 1990.
4. N. Choquet. Test data generation using a prolog with constraints. In *Proc. of the Workshop on Software Testing*, pages 132–141. Computer Science Press, Banff Canada, July 1986.
5. W.M. Craft. *Detecting Equivalents Mutants Using Compiler Optimization.* Master Thesis, Department of Computer Science, Clemson University, Clemson-SC, 1989.
6. R.A. De Millo, D.C. Gwind, and K.N. King. An extented 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.
7. 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.
8. M. E. Delamaro and et al. Integration testing using interface mutation. In *VII International Symposium of Software Reliability Engineering (ISSRE)*, pages 112–121. IEEE Computer Society Press, New York, NY, November 1996.
9. M. E. Delamaro and J.C. Maldonado. A tool for the assesment fo 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.
10. R. Denney. Test case generation form prolog-based specifications. *IEEE Software*, pages 49–57, March 1991.
11. M.C.F.P. Emer and S.R. Vergilio. Selection and evaluation of test data sets based on genetic programming. *Software Quality Journal*, pages 167–186, June 2003.
12. F.G. Frankl and E.J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the Workshop on Software Testing*, pages 4–13. Computer Science Press, Banff - Canada, July 1986.
13. M.M. Gorlick, C.F. Kesselman, D.A. Marotta, and S Parker. Mockingbird: logical methodology for testing. *Journal of Logic Programming*, (8):95–119, 1990.
14. D. Hoffman and P. Strooper. Automated module testing in prolog. *iEEE Transactions on Software Engineering*, 17(9):934–943, 1991.
15. G.B. Luo, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. pages 104–113, March 1992.
16. A.P Mathur and W.E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *The Journal of Software Testing, Verification Reliability*, Vol. 4(1):9–31, March 1994.
17. L.M. Pereira. Rational debugging in logic programming. In *Third International Conference on Logic Programming*, pages 203–210. Lectures Notes on Computer Science, 1986.
18. L. Plumer. Termination proofs for logic programs. In *Lectures Notes in Artificial Intelligence*. Spring-Verlag, 1990.
19. S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.
20. W.E. Wong. *On Mutation and Data Flow.* PhD Thesis, Department of Computer Science, Purdue University, West Lafayette-IN, USA, December 1993.