Muta-Pro: Towards the Definition of a Mutation Testing Process

A. M. R. Vincenzi[*1], A. S. Simão[◇], M. E. Delamaro[†] & J. C. Maldonado[◇]

[*]Instituto de Informática
Universidade Federal de Goiás
Goiânia, GO, Brazil
auri@inf.ufg.br

[◇]Inst. de Ciências Mat. e de Computação
Universidade de São Paulo
São Carlos, SP, Brazil
{*adenilso, jcmaldon*}*@icmc.usp.br*

[†]Centro Universitário Eurípides de Marília
Marília, São Paulo, Brazil
delamaro@fundanet.br

Abstract

Mutation Testing originated from a classical method for digital circuit testing and today is used at program and specification levels. It can be used either to generate or to assess the quality of test sets. In spite of being very effective in detecting faults, Mutation Testing is usually considered a high cost criterion due to: i) the large number of generated mutants; ii) the time-consuming activity of determining equivalent mutants; and iii) the mutant execution time. Many initiatives aiming at reducing the Mutation Testing application cost have been conducted, most of them addressing one of the drawbacks mentioned above.

In this paper, we identify and summarize some of the most relevant researches and results related to Mutation Testing cost reduction, e.g., Constrained-Mutation, Constraint-Based Testing and Bayesian Learning. Moreover, we propose a Mutation Testing process, named Muta-Pro*, that synergetically integrates the related approaches and mechanisms. This process is intended to be incremental and tailorable to a specific application domain such as C programs or finite state machine models. The main ideas in this paper are illustrated using a UNIX utility program.*

This process is being integrated in a Mutation Testing environment, based on the authors' previous experience on implementing the Proteum Family *tools, aiming at promoting the technology transfer to industry and providing the basis for improving the* Muta-Pro *process itself.*

Keywords: Mutation Testing, Mutation Testing Process, Testing Environment.

1. Introduction

Testing may be considered an incremental activity that pervades most, if not all, of the software development cycle. The success of the testing activity depends on the quality of a test set. Since the exhaustive test is, in general, impracticable, criteria that allow selecting a subset of the input domain by preserving the probability of revealing the existent faults in the program are necessary. There is a large number of criteria available to evaluate a test set for a given program against a given specification (see Zhu *et al.* [51] for a survey). The idea behind a testing criteria is to systematize the testing activity, as well as to provide a coverage measure. Testing criteria are usually classified in functional [39], structural [41] or fault-based [13] techniques, according to the source of information used to extract the testing requirements.

[1]Auri Marcelo Rizzo Vincenzi
Bloco IMF I, sala 239 - Campus II - Samambaia
Caixa Postal 131, 74001-970
Goiânia, GO, Brazil
Tel: +55 62 521-1181 – Fax: +55 62 521-1182
e-mail: auri@inf.ufg.br

One important point to be highlighted is that testing criteria and techniques are complementary and the tester may use one or more of them to assess the adequacy of a test set with respect to (w.r.t.) a program and to eventually enhance the test set by devising additional test cases needed to satisfy the selected criteria.

Mutation Testing appeared in the 70's and was strongly influenced by a classical method for digital circuit testing known as "single fault test model" [18]. It has been used to test different products, including programs [13, 11, 5] and specifications [16, 17, 45, 40, 46].

Mutation Testing requires the development of a test set T that may reveal the presence of a well-specified set of faults [13]. The faults are modeled by a set of mutant operators which, when applied to a program P under test, generate syntactically correct programs, called mutants, with minor differences w.r.t. P. The quality of T is measured by its ability to distinguish the behavior of the mutants from the behavior of the original program. In spite of its effectiveness [11], Mutation Testing possesses drawbacks, that makes its practical application difficult, such as:

- the large number of mutants that are generated;
- the need to inspect many mutants and analyze them for possible equivalence w.r.t. P; and
- the time consuming activity of executing the mutants against the test set.

As an attempt to reduce its cost, some approaches derived from Mutation Testing, referred by us as Alternative Criteria, have been investigated, e.g., Randomly Selected Mutation [1], Constrained Mutation [29] and Selective Mutation [35, 32, 4]. These approaches try to determine a subset of mutants in such a way that, if a test set T is able to distinguish such mutants, then T would also distinguish the complete set of mutants. However, the task of determining equivalent mutants remains.

With respect to the task of determining equivalent mutants, some initiatives can also be identified. Since it is not a trivial problem, different techniques have been developed and investigated in this context, trying either to automate the determination of equivalent mutants or to provide guidelines to ease this task, e.g Dependence Analysis [20], Amorphous Slicing [21], Constraints [14], Compiler Optimizations Techniques [33, 36, 37], and Artificial Intelligence Techniques [50].

Considering the quantity of researches related to Mutation Testing, it is important to think of a mutation testing process that aggregates these techniques aiming at providing an effective, low-cost and practical way of applying Mutation Testing. In a previous work, Harman *et al.* [20] defined a mutation testing process that used Dependence Analysis [22] and Constraints [14]. The former was used to check for equivalent mutants and the latter was used for both test case generation and equivalent mutant determination. The idea was to use this technique to reduce the human effort to carry out these activities.

In this paper, we aim at combining some of these researches in order to define a mutation testing process, named *Muta-Pro*. *Muta-Pro* should be flexible in the sense that it can be instantiated according to the objective, time and cost constraints of a given product, such as high-level hardware specifications, formal specifications or programs. Considering this objective, in Section 2 we present the background information which is necessary to understand the Mutation Testing criterion and the four steps of its application. In Section 3 we present some related work classifying them according to each Mutation Testing step. In Section 4 we define *Muta-Pro* by pointing out where each work can be applied. In Section 4.1 we present an example of the *Muta-Pro* instantiation, showing some data collected using an UNIX utility program. In Section 5 we present the conclusions and future work related to this research.

2. MUTATION TESTING

In this section, we briefly explain the main concepts of Mutation Testing which are necessary to understand this paper. Mutation Testing provides the tester with a systematic way both to evaluate how "good" a test set is and to generate test cases. Let S be a specification and P be a program that supposedly implements S. Let D be the input domain of P and $T \subset D$ be a test set for P. Basically, the application of Mutation Testing consists of four steps: Mutant Generation, Program Execution, Mutant Execution, and Mutant Analysis. These steps are represented in Figure 1, by use of a Structured Analysis and Design Technique (SADT) diagram [42].

SADT basic build block is a box whose sides represents Input, Control, Output and Mechanism, respectively. Boxes are connected by arrows which go from an Output of one box to the Input or Control of another box. As stated by Ross [42], the names **Input** and **Output** are chosen to convey the idea that the box represents a *transformation* from a previous to a succeeding state of affairs. The **Control** interface *interacts* and *constrains* the transformation to ensure that it applies only under the appropriate circumstances. Therefore, the combination of Input, Output and Control fully specifies the bounded piece of subject, and the interfaces relate it to the other pieces. Finally, the **Mechanism** support *provides means* for carrying out the complete piece represented by the box.

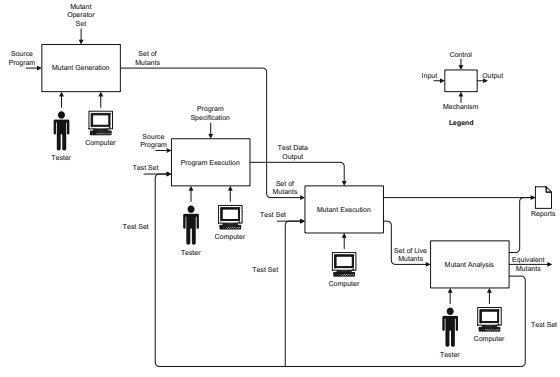


Figure 1. Steps of Mutation Testing.

Mutant Generation The mutant generation phase is one of the most important ones, since the mutants are responsible for either evaluating the quality of a given test set or generating a test set.

A set of alternative implementations containing simple syntactic changes is created. The syntactic changes are modeled by **mutant operators** which can be thought of as a **fault model** that corresponds to the common faults committed during software development. In general, the more mutant operators are used, the more mutants are generated, which increases the cost of Mutation Testing. Mutant operators depend on the language in which the artifact to be tested is described.

Program Execution In this step, P is executed against the test set T. If there exists $t \in T$ such that $P(t) \neq S(t)$ a fault was discovered[1], the Mutation Testing finishes and P should be corrected. If for every $t \in T$, $P(t) = S(t)$, it is said that P corresponds to S w.r.t. T. A question in this case is whether P is correct or T is of low-quality. Observe that in this step, the existence of an oracle (in general the tester) is necessary to decide whether $P(t) = S(t)$, for each $t \in T$.

Mutant Execution If T does not reveal any fault in P, the quality of T is evaluated in this step. Let M be the set of mutants generated in first step. For each $m \in M$ and each $t \in T$, $m(t)$ is compared with $P(t)$. If there is a $t \in T$ such that $m(t) \neq P(t)$, m is said to be "killed" by t. On the other hand, when $m(t) = P(t)$ for every $t \in T$, m is said to be "alive" and should be analyzed in the next step.

After this step, the mutation score is calculated. The **mutation score** is the ratio of the number of dead mutants to the number of non-equivalent mutants and provides the tester with a mechanism to assess the quality of the testing activity. In this way, the mutation score can be used to evaluate if the testing objective was reached. In the case that the mutation score reaches 1.00 (100%), it is said that the T is adequate w.r.t. Mutation Testing (**MT-adequate**) to test P. Since the identification of equivalent mutants is important to compute the mutation score, usually, a value of 1.00 is obtained only after the analysis of the live mutants.

Mutant Analysis A mutant m can stay alive after being executed with T either because m is equivalent to P ($m \equiv P$) and will always behave identically to P, for all $t \in D$; or because T is not good enough to show the difference in the behavior of m and P, and there exists a $t' \in (D \backslash T)$ such that $m(t') \neq P(t')$. If the mutant is equivalent, it can be dropped out. Otherwise the tester could develop new test data to show the difference in the behavior of the alive mutant m and P.

As stated by Ghosh and Mathur [19], a tester is expected to kill each mutant in M with at least one test case t. In case a mutant cannot be killed, the tester needs to show that $m \equiv P$.

There are different ways to compare the behavior of a program P and a mutant m. Considering *strong mutation* – traditional Mutation Testing – P and m are executed and, at the end of execution, the outputs are compared to determine whether they behaves differently. In *weak mutation* P and m are compared with the values of some variables just after the mutated statement in m.

Let s_i be the statement in P that has been mutated to s_i' to obtain m. DeMillo and Offutt [14] defined tree conditions that a test case t must satisfy in order to distinguish m from P.

1. **Reachability**: when executing m with t, the control must reach s_i';
2. **Necessity**: the state of m immediately following some execution of s_i' must be different from the state of P immediately following the corresponding execution of s_i;
3. **Sufficiency**: the difference in the state of P and m immediately following the execution of s_i and s_i' must propagate to the end of execution of P or m such that $P(t) \neq m(t)$.

In case of *weak mutation* the two first conditions (Reachability and Necessity) are enough to kill the mutant, but *strong mutation* requires the three conditions.

[1] We use the notation $P(t)$ to represent the output of P when executed against a given test case t.

It must be pointed out that applying the testing criteria without the support of a tool is an error-prone and unproductive activity. The availability of a testing tool increases the quality and the productivity of the test activity and may ease the technology transfer to the industry, contributing to the continuous evolution of such environment, indispensable for the production of high quality software products.

Considering Mutation Testing, the first tool to support the testing of C programs based on mutation testing at unit level was *Proteum* [10]. With the proposition of the criterion Interface Mutation [11], that uses a set of mutant operators developed to model integration faults, the PROTEUM/IM has been developed [9]. Recently, *Proteum* and PROTEUM/IM have been integrated in a testing environment, named PROTEUM/IM 2.0 [12]. In this way, the tester can use the same concept during the unit and the integration testing phases. This paper uses PROTEUM/IM 2.0 to provide support to *Muta-Pro*. In the next section we discuss the support provided by PROTEUM/IM 2.0 according to each step of Mutation Testing and also other related works which are useful to reduce the cost of Mutation Testing.

3. Previous Work

This section presents some researches related to the cost reduction of Mutation Testing. We organize the section according to the main steps of Mutation Testing. Although there is a lot of researches related to Mutation Testing, we give special attention to those developed by our research group during these last years.

3.1. On Mutant Generation

So far mutation testing has been applied to programs written in several programming languages, like Fortran [14], C [2] and Java [27, 5], and to formal specifications written using Finite State Machines [16], Petri Nets [17], Statecharts [45], Estelle [40] and SDL [46]. This flexibility derives from the fact that mutation testing requires only an executable model that transforms inputs into observable outputs that can be compared against the results produced by the mutants.

Mutant operators are designed by referring to the experience of using the target language and of the most common faults. In the past, mutant operators were designed based on experts' knowledge. Recently, Kim *et al.* [26] have proposed the use of a technique named "Hazard and Operability Studies" (HAZOP) to systematically derive mutant operators, illustrated using the Java language. Although the resulting operators do not significantly differ from past works, the proposed methodology is an important step towards a more rigorous discipline in the creation of mutant operators. The technique is based on two main concepts. It first identifies in the grammar of the target language the points that are subject to mutation and then, based on suitable "Guide Words", defines the set of mutations that can occur in these points.

The mutant operators implementation is a time consuming activity in itself. This fact motivated the proposition of the language $\mathcal{MUDEL}$ (standing for **Mu**tant **De**scription **L**anguage), aiming not only at automating the mutant generation, but also at providing precision and formality to the operator descriptions [43]. To support $\mathcal{MUDEL}$, an application generator, named `mudelgen` (standing for $\mathcal{MUDEL}$ Generator), has been developed. These mechanisms have been applied for C and Petri-Nets and currently been investigated for defining C++ and Java mutant operators.

The cost and efficacy of Mutation Testing is related to the cost and efficacy of the defined mutant operator set. Thus, one way to reduce its cost is to use subsets of operators or mutants that would lead to the selection of test sets as effective as the total set of operators and mutants would [35, 32, 4].

Using a sufficient mutant operators set, it is expected to achieve a high cost reduction, considering the number of mutants which are generated, as well as a high mutation score w.r.t. the total set of operators [35].

Considering the determination of a sufficient mutation operators set, Barbosa *et al.* [4] developed a procedure, named *Sufficient*, that provides a systematic way to select a subset of operators for C language at the unit level. Using two different program suites, Barbosa *et al.* determined the sufficient mutant operators set. The sufficient set provides a high adequacy degree w.r.t. Mutation Analysis with a high cost reduction in terms of the number of mutants. Moreover, given the application cost and the test requirements that each mutation class determines, the *Sufficient* procedure establishes an incremental strategy of application among the mutant operators of the sufficient set. The idea is to first apply the mutant operators that are relevant to certain minimal requirements of testing (e.g., all statements execution, all control flow coverage). Next, depending on the criticality of the application and on the budget and time constraints, the mutant operators related to other concepts and test requirements may be applied [4].

Motivated by the development of an incremental testing strategy for applying mutant operators, Vincenzi *et al.* [49] evaluated the 71 mutant operators implemented in the *Proteum* [10] and the 33 mutant operators implemented in PROTEUM/IM [12] testing tools. These tools support the application of Mutation Testing at unit and integration level for C programs, respectively. Currently these tools are integrated in a single environment named PROTEUM/IM 2.0 [12]. At the unit level, Mutation Testing

is also refereed to as Mutation Analysis (MA) [13] and at the integration level as Interface Mutation (IM) [11]. When no distinction is necessary we will employ the term Mutation Testing. Vincenzi *et al.* [49] conducted an experiment divided in two phases. In the first phase, the Mutation Analysis and Interface Mutation criteria were analyzed one at a time. The idea was to minimize the cost of application of these criteria in case they had to be separately applied, aiming at establishing an incremental way to apply them in isolation. In the second phase, the results from the first phase were combined, considering that an IM-adequate test set would have to be evolved from an MA-adequate test set. The best results were obtained when the sufficient mutant operators sets [4] were applied first in the strategy, originating the Sufficient Incremental Unit Testing Strategy (*SUS*) [49].

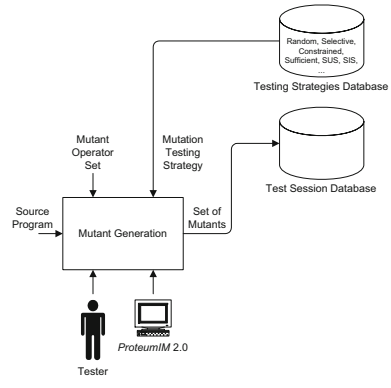
Using the*SUS*, it was possible to obtain an MA-adequate test set using 20 out of the 71 unit operators with a cost reduction of 40% w.r.t. Mutation Analysis. Using only 5 out of 20 operators (the most prevalent ones), it was possible to obtain test sets that determine a mutation score of 0.995 w.r.t. Mutation Analysis, with a cost reduction near to 87%, on average.

Another point observed by Vincenzi *et al.* [49] is that to improve a mutation score from 0.995 to 1.000, the cost reduction (in terms of number of mutants) decreases more than 45%. In this way, the idea is to prioritize the application of some mutant operators based on some aspect that the tester would like to highlight, considering the time and cost constraints. Using an incremental strategy, the tester reduces the complexity of executing and evaluating significative amount of mutants.

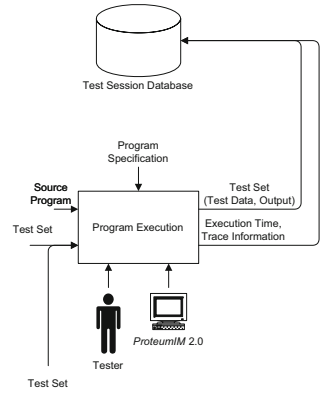
The same evaluation was done considering the Interface Mutation mutant operators. Considering the Sufficient Incremental Interface Testing Strategy (*SIS*), using 18 out of the 33 interface operators, it was possible to obtain an IM-adequate test set with a cost reduction around 26%. With only the 6 prevalent ones a mutation score of 0.994 and a cost reduction close to 88% were obtained, on average [49].

In the development of these strategies, Vincenzi *et al.* [49] group the mutant operators prioritizing the application of the mutant operators that provide the highest mutation score w.r.t. the overall set of mutant operators with the lowest cost, in terms of the number of generated mutants. Other incremental strategies have also been developed considering the cost in terms of the number of equivalent mutants. In this case, the goal is to obtain a higher mutations score with a lower cost in analyzing equivalent mutants [24].

Figure 2(a) illustrates which are the controls that can be applied to the first step to reduce the cost of Mutation Testing in terms of the number of generated mutants. *PROTEUM/IM* 2.0, as *Proteum* and *PROTEUM/IM*, uses the concept of **test session**. Therefore, a test session is characterized by a database that is created and managed by the tools to store the necessary information about the program under testing, i.e., its mutants and test cases. It can also be observed that $\mathcal{MuDeL}$ can be used in this phase to provide a new kind of mutant operator which is specific for a given program or application domain.



(a) Step 1



(b) Step 2

Figure 2. *Muta-Pro*: Improvements on Mutant Generation and Program Execution.

3.2. On Program Execution

During the program execution, if desired, *PROTEUM/IM* 2.0 instruments the source program aiming at collecting control flow information to accelerate the mutant execution. When a test case *t* is included in the test set, the tool stores information about the nodes of the program graph that were reached by its execution on the program under testing and also the time spent on each test case execution.

To register control flow information it is necessary to instrument the source program such that, when executing it with each test case, the respective execution path could be collected and stored. Using the same idea of

$\mathcal{MuDeL}$, Simão *et al.* [44] have also been developed an *Instrumentation Description Language*, named IDeL, to support this activity. Similarly to `mudelgen`, Simão *et al.* developed `idelgen` to support IDeL application. More information about IDeL and `idelgen` can be found elsewhere [44].

Figure 2(b) gives a general idea about this step of *Muta-Pro*. Given P and T, PROTEUM/IM 2.0 executes each test data with the original program, grabs and stores the test data output, the time of execution and the trace information. An oracle, generally the tester, should decide whether $P(t) = S(t)$, $t \in T$. If exists $t \in T$ such that $P(t) \neq S(t)$, the process finishes and P should be corrected because a fault was discovered.

3.3. On Mutant Execution

The execution of mutants is one of the bottlenecks for mutation testing, and some approaches can be used to speed up this step. Considering PROTEUM/IM 2.0 [12], three approaches were developed:

1. compilation (creation of executable mutants) instead of interpretation;
2. time reduction to create the mutant sources; and
3. storage of control-flow information to accelerate the mutant execution.

The compilation approach contributes to improve the execution of mutants but, on the other hand, introduces a delay in building mutant sources and executables. To reduce this delay, PROTEUM/IM 2.0 builds source files that hold several mutants at once. This approach has been addressed by many authors [25, 48]. The process of construction/compilation is not carried out for each mutant. Instead, it is done once for a "large" number of mutants (currently 100 at most).

PROTEUM/IM 2.0 also uses control flow information to accelerate the mutant execution. When a test case t is included in the test set, the tool stores information about the nodes of the program graph that were reached by its execution on the program under testing, i.e. the tool checks the reachability condition of t related to a specific mutant m. Before trying to kill m using the test case t, PROTEUM/IM 2.0 checks if t really can kill m, i.e., it checks whether the node(s) in which the mutation was done in the original program is reached by t. In general a large number of executions is avoided and the execution process is significantly improved [12].

Figure 3 presents the idea of this step of *Muta-Pro*. At first, the set of mutants, the set of test data, and the output and execution time of each test case are recovered from the Test Session Database (TSD). PROTEUM/IM 2.0 is responsible for deciding which mutants should be executed with each test data based on control flow information, and then for joining and compiling them. When the execution time of a given mutant is a certain number of times greater than the execution previously recorded for the current test data, the tool automatically kills this mutant assuming that it is in infinite looping. At the end, the sets of live and dead mutants are produced and stored into TSD. Observe that this kind of information is also useful for developing new testing strategies, trying to avoid the generation of mutants that die easily [24].

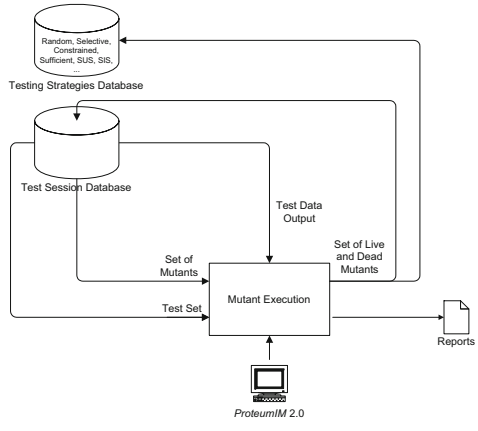


Figure 3. *Muta-Pro*: Improvements on Mutant Execution - Step 3.

3.4. On Mutant Analysis

During the mutant analysis, two tasks should be accomplished: the determination of equivalent mutants and the generation of new test cases to kill a given mutant. Unfortunately, both tasks are difficult to be automated. In general, it is undecidable whether there exists an input data that causes for a given path in a program to be traversed. Moreover, this limitation impacts the automatic generation of test cases. It is also undecidable whether two programs (P and P') compute the same function (are equivalent), i.e., $P(t) = P'(t), \forall t \in D$.

Detecting Equivalent Mutants Several approaches have considered the problems of equivalent mutant determination, using both constraint-based techniques and compiler optimizations [14, 34, 36, 37]. The idea explored by Offutt and Craft [33] was to implement a set of compiler-optimization heuristics and evaluate them. The approach consists of looking at the mutants which, compared to the original program, implement traditional peep-hole compiler optimizations [3]. Compiler optimizations are designed to create faster but equivalent programs, so a mutant which implements a compiler optimization is, by definition, an equivalent mutant. The set of

implemented heuristics was able to detect about 10% of the equivalent mutants.

Another study developed by Harman *et al.* [20] explores the relationship between program dependence and mutation testing. The idea was to combine dependence analysis tools with existing mutation testing tools, supporting the test data generation and the determination of equivalent mutants. The authors also proposed a new mutation testing process which starts and ends with dependence analysis phases. In the first phase of the process the mutants are created using an independent approach. Next, the process tries to applies dependence analysis [22] and uses the constraint-based approach [14] for both generating test cases to kill some mutants and determining others to be equivalent. The mutants that are alive after these steps – called *stubborn mutants* – are those that are not determined to be equivalent by neither dependence analysis nor by constraint-based analysis.

Harman *et al.* suggest that the *stubborn mutants* will ultimately have to be considered by a human, but before the human analysis occurs, two more phases take place:

1. amorphous slicing – that produces a simplified program tailored to the question whether or not the mutant is equivalent [21]; and
2. domain reduction – obtained by using dependence analysis [22].

Vincenzi *et al.* [50] developed an approach, named Bayesian Learning-Based Equivalent Detection Technique (BaLBEDeT), that uses Bayesian Learning [31] to estimate which is the most promising group of mutants that should be analyzed, considering a certain number of test cases. Each mutant operator has specific characteristics, i.e., one mutant operator may generate more equivalent mutants than another one. Based on these characteristics and on historical information previously collected [4, 49], the Brute-Force algorithm [31] (based on Bayesian Learning) is used to guide the analysis of the live mutants, aiming at reducing the effort to determine the equivalent ones. The idea is to provide guidelines to ease the analysis of the live mutants.

Given the number of test cases which were executed, the technique indicates, for each mutant operator *op*, the probability of the live mutants of *op* to be equivalent or non-equivalent. Considering L_{op} the number of live mutants generated by *op*, E_{op} is the probability of L_{op} to be equivalent, and NE_{op} is the probability of L_{op} to be non-equivalent. If L_{op} is approximately equal E_{op} ($L_{op} \approx E_{op}$), the mutants of *op* should be dismissed because most of them are equivalent. On the other hand, if $L_{op} \approx NE_{op}$ the mutants *op* should be analyzed because there is a high probability that the live mutants will be killed. Based on this information the tester can decide which mutants should be analyzed first.

Table 1 illustrates the kind of information provided by BaLBEDeT, considering the set of mutant operators for C language. For example, considering a test set with 20 elements, the probability of u-Cccr's live mutants to be equivalent ($P(\oplus|$u-Cccr$)$) is 0,40 against 0,60 to be non-equivalent ($P(\ominus|$u-Cccr$)$). For a test set with 100 elements, these probabilities are 0,75 and 0,25, respectively.

Table 1. BaLBEDeT's Probabilities: (a) 20 test cases, (b) 100 test cases

	(a)		(b)	
Operator	$P(\ominus\|op)$	$P(\oplus\|op)$	$P(\ominus\|op)$	$P(\oplus\|op)$
u-Cccr	0,60	0,40	0,25	0,75
u-Ccsr	0,98	0,02	0,93	0,07
...	...	...	...	...
u-OLBN	0,28	0,72	0,11	0,89
u-ORRN	0,58	0,42	0,32	0,68
...	...	...	...	...
u-SCRB	0,00	1,00	0,00	1,00
u-SSDL	0,70	0,30	0,34	0,66
...	...	...	...	...
u-VDTR	0,14	0,86	0,04	0,96
u-VTWD	0,73	0,27	0,41	0,59

Test Case Generation The task of automatically generating test cases can be divided into three classes: random, static and dynamic.

Random test data generation (e.g. [15]) is easy to automate, but may create many test data and may fail to find test data to satisfy a given requirement because information about the test requirement is, in general, not incorporated into the generation process.

Static generation does not require the program execution. One static technique is symbolic execution. It works by traversing a control flow graph of the program and building up symbolic representations of the internal variables in terms of the input variables, for the desired path [8, 14]. Branches within the code introduce constraints on the variables. Solutions to these constraints represent the desired test data. A number of problems exist with this approach. By using symbolic execution it is difficult to analyze recursion, array indexes which depend on input data and some loop structures. Also, the problem of solving arbitrary constraints is known to be undecidable [47].

Dynamic test data generation involves the execution of the program and a directed search for test

data that meets the desired criterion. The work of Korel [28] uses locally directed search techniques, but these techniques only work effectively for linear continuous functions and are likely to become stuck at a local optimum and fail to locate the required global optimum [23]. The use of global optimization techniques for dynamic test data generation has been investigated more recently in an attempt to overcome this limitation [23, 7, 6].

Considering Mutation Testing, DeMillo and Offutt [14], using the concept of constraint, developed an automatic way to generate test cases. The idea was that, by solving a set of constraint, it is possible to generate a test case that kills a given mutant. Even not being satisfied, the set of constraint is also useful to determine equivalent mutants. Empirical studies showed that the approach could achieve a detection rate of equivalent mutants around 50% [36, 37]. The use of artificial Genetic Algorithms (GAs) and other artificial intelligence search techniques have also been explored as one alternative to test case generation. There has been a significant amount of research in this area, mainly related to control and data-flow based criteria [30, 23, 38, 47, 7].

The crucial choice to be made when using GAs is the definition of the fitness function, responsible for classifying the best individuals (test cases). For example, considering the problem of test cases generation to kill a given mutant, the fitness function should consider the reachability, necessity and sufficiency conditions [14]. Based on these three conditions, Bottaci [6] developed a genetic algorithm fitness function for Mutation Testing to automatically and effectively generate test cases to kill a given mutant. As stated by Bottaci, the empirical investigation of the proposed fitness function was not carried out and should be done. Anyway, research in this area w.r.t. Mutation Testing is in an early stage and we are also working on this direction.

Figure 4 depicts the general idea of how to improve the Mutation Analysis step. As can be observed, from a given set of live mutants, the tester can use some guidelines or heuristics to ease the identification of equivalent/non-equivalent mutants, based on previously data stored on a Knowledge Database. After determining the equivalent ones, the tester should generate a testing report to evaluate the quality of *T*. If the coverage is not the desired one, more test data should be generated to kill the live non-equivalent mutants and, with the new test set, return to Step 2 (Program Execution).

Observe that, each time equivalent mutants are determined for a new program under testing, both Testing Strategies Database and Knowledge Database are updated. The idea is to continuously update these databases to improve the incremental testing strategies and also to ease, as soon as possible, the analysis of live mutants.

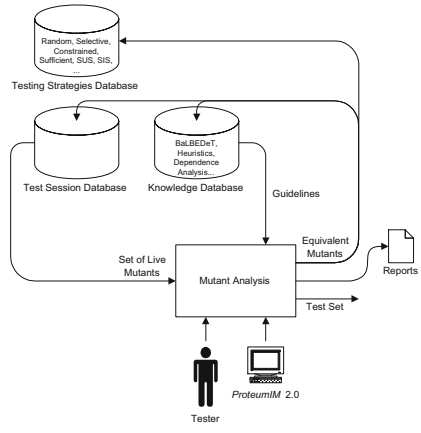


Figure 4. *Muta-Pro*: Improvements on Mutant Analysis - Step 4.

4. Mutation Testing Process

Motivated by Harman *et al.* [20], described in Section 3.4, a mutation testing process (*Muta-Pro*) is presented below that intends not only to use dependence analysis and constraint-based approaches to ease the application of Mutation Testing, but also that aggregates our previous experience in the development of mutation testing strategies, tools and guidelines to analyze live mutants. Observe that, although *Muta-Pro* is defined based on previous works carried out at program level, the ideas presented herein can easily be extended to other contexts where mutation testing is also applied.

The steps of application of Mutation Testing are described as a flow chart illustrated in Figure 5.

4.1. Muta-Pro Instantiation

Let *P* be the program under testing, *OP* be the total set of mutant operators, and *SQ* be the sequence of mutant operators to be applied incrementally ($SQ = \langle op_1, op_2, \ldots, op_n \rangle$ such that $op_i \in OP$ for $1 \leq i \leq n$). Using the *Comm* UNIX utility program, we present how the *Muta-Pro* can be instantiated and the results obtained by applying it. Again, the process is illustrated considering the steps of Mutation Testing application.

Comm, which has 119 LOC (Lines Of Code), compares two sorted files line by line, producing a three-column output: column one contains lines unique to the first file, column two contains lines unique to the second file, and column three contains the lines common to both files.

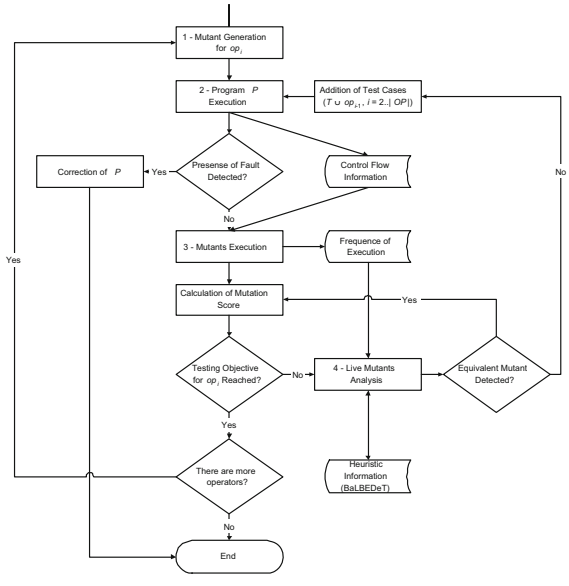


Figure 5. Mutation Testing Process – *Muta-Pro*.

Mutant Generation

Mutant generation should be an incremental activity, trying to reduce the complexity of Mutation Testing. Considering the set of all unit mutant operators implemented in the *PROTEUM/IM* 2.0 testing tool, we instantiate the sequence of application of mutant operators as the one defined by the Sufficient Incremental Unit Testing Strategy– *SUS* [49], i.e. $SQ = \langle$u-SMTC, u-SSDL, u-OEBA, u-ORRN, u-VTWD, u-VDTR, u-OBSN, u-OASN, u-OLRN, u-SWDD, u-VLAR, u-VGAR, u-Oido, u-SSWM, u-OEAA, u-ORBN, u-SRSR, u-STRI, u-VLSR, u-VGSR, u-OABN, u-Cccr$\rangle$. In this way, considering the source program *Comm*, *PROTEUM/IM* 2.0 generates and stores the set of mutants for each one of the *SUS*'s mutant operators. Figure 6 illustrates the cost of each SUS's mutant operator in terms of the number of generated mutants.

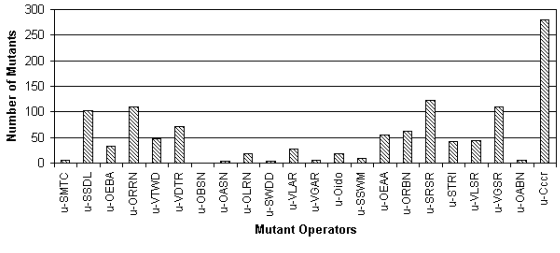


Figure 6. Cost of each SUS's mutant operator.

Figure 7 illustrates the cost of *SUS* w.r.t. *OP* (the total set of mutant operators). Considering *OP*, 1,632 mutants are generated for *Comm* program. *SUS* Essential (first six operators of SQ) is responsible by generating 23% of the mutants, SUS Non-Essential (other operators of SQ) by 49% and the other mutants operators (the ones not used by *SUS*) by 28%.

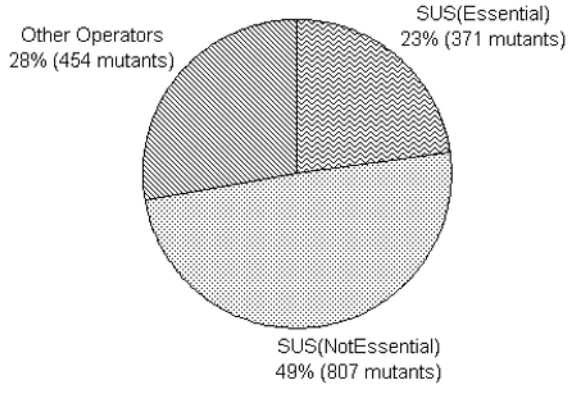


Figure 7. Cost of each SUS w.r.t. Mutation Analysis.

Program Execution

In this step, *PROTEUM/IM* 2.0 registers the execution path of each test case $t \in T$ when executing it against *P*. Therefore, this information can be used to reduce the time spent during the mutants execution. In order to collect this information, *PROTEUM/IM* 2.0 requires that *P* be instrumented. Considering the *Comm* program (119 LOC), the instrumentation takes around 230 milliseconds which can be considered very low compared to the gain that is obtained during the mutant execution which is shown in the following section.

Mutants Execution

In this step, some approaches are used to accelerate as much as possible the time expended during the mutant execution. In the case of *PROTEUM/IM* 2.0 the use of compilative approach, the creation of multiple mutants in a single source code and the use of control-flow information to avoid the execution with some test cases are some of these approaches.

Figure 8 illustrates the difference in the execution time of the mutants considering an instrumented code (that enables the collection of control-flow information) and a non-instrumented one. Observe that the instrumented code provides a faster mechanism to speed up the mutant execution time and the more mutants are generated, the greater the time difference is.

Another important information derived from the control-flow information is related to the **frequency of execution** of each mutant. Since *PROTEUM/IM* 2.0 registers whether a test case executes or not a given mutated object, for each mutant there is information about how many test cases in the test set touch the mutated part,

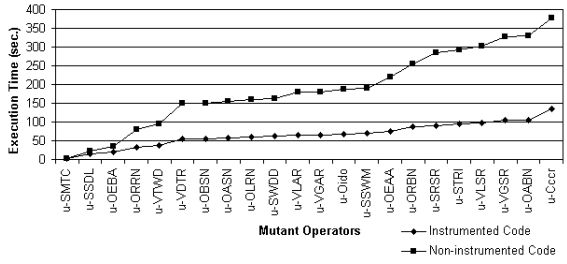


Figure 8. Mutant Execution Time: Instrumented *versus* Non-Instrumented Code.

i.e. satisfied the reachability condition. The frequency of execution combined with some probabilistic information about each mutant operator can be used to automatically determine equivalent mutants.

Mutants Analysis

After the execution of the mutants, the ones that survive should be analyzed either to improve the test set or to determine the equivalent ones. In this step, BaLBEDeT and the frequency of execution can be used to provide an automatic way to determine equivalent mutants. Therefore, considering the set of live mutants (LM_i), the current test set (T_{i-1}) and the current mutant operator op_i, BaLBEDeT takes the cardinality of T_{i-1} and provides the probability of op_i's live mutants to be equivalent ($P(\oplus|op_i)$). The set of op_i's live mutants is classified according to the frequency of execution and the most prevalent ones are marked as equivalent until the probability $P(\oplus|op_i)$ be reached. Figure 9 shows the real number of equivalent mutants for each *SUS* mutant operator and the estimated one.

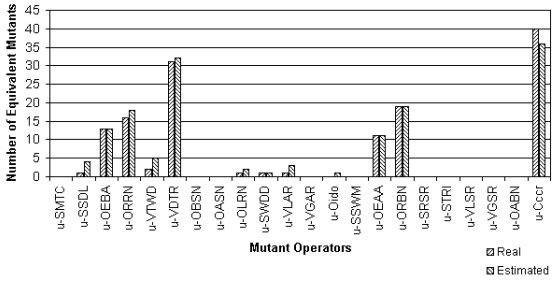


Figure 9. Determination of Equivalents Mutants: BaLBEDeT and *frequency of execution*.

As can be observed, BaLBEDeT overestimates the number of equivalent mutants for six mutant operators and underestimates one. We consider a reasonable result since BaLBEDeT provided the probabilistic information based on previous data w.r.t. equivalent mutants, so a more accurate result will be obtained as more information about equivalent mutants is collected in other programs.

After eliminating the equivalent mutants, the live ones are expected to be killed by one or more test cases. In this version of *Muta-Pro*, this activity was performed by hand, i.e. the tester has to analyze the live mutants and generate one or more test cases to kill them.

To compare the performance of the *SUS* mutant operators with the Mutation Analysis criterion, at each step of the *Muta-Pro* algorithm, we evaluate the adequacy of T_i against *OP* and the associated cost. Figure 10 shows the obtained data.

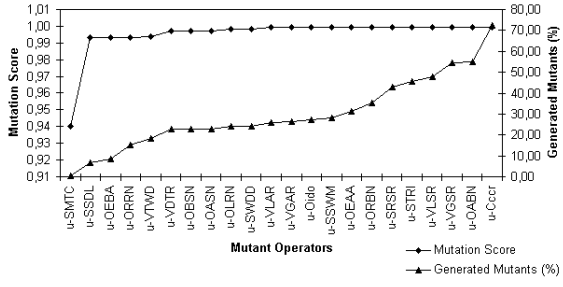


Figure 10. *SUS versus* Mutation Analysis: Mutation Score and Cost.

Observe that from a previous functional test set T_0, applying the second mutant operator (u-SSDL), a mutation score above 0.99 w.r.t. Mutation Analysis is obtained with a cost around 93%, since u-SMTC and u-SSDL are responsible for generating less then 7% of the total of mutants. We can also observe that for u-VDTR mutant operator to obtain a little increment in the mutation score it is necessary to execute and evaluate a greater number of mutants which motivates the improvement and the development of more accurate techniques for both the generation of test cases and the determination of equivalent mutants.

5. CONCLUSIONS

In this paper, we proposed a mutation testing process, *Muta-Pro*, that integrates research under development in our research group with others from the literature. *Muta-Pro* provides an approach to apply mutation testing incrementally, considering the time and cost constraints.

We are working on integrating *Muta-Pro* into *PROTEUM/IM* 2.0 testing environment. This integration will allow us to conduct empirical studies evaluating the mutation testing process and the relationship between Mutation Testing and other criteria. The main objective which has been pursued is the definition of an integrated testing environment that enable us to apply mutation based testing criteria in a low-cost and effective way, such that they can be used by the industry in the improvement of their products.

Another research interest is to use Genetic Algorithms to try to improve some parts of the *Muta-Pro*, mainly w.r.t. the automatic determination of equivalent mutants and to generate test cases to kill a mutant. The most important work in this sense is the definition of a fitness function that satisfies the three conditions required to kill a mutant: reachability, necessity and sufficient conditions. There are different approaches that can be used to associate a given cost with each one of these conditions and we are currently investigating such approaches.

6. Acknowledgments

The authors would like to thank the Brazilian Funding Agencies – CNPq, FAPESP, CAPES and FUNAPE – and the Telcordia Technologies (USA) for their partial support to this research.

References

[1] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[2] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, Mar. 1989.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1996.

[4] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Towards the determination of sufficient mutant operators for C. *STVR – Software Testing, Verification and Reliability*, 11(2):113–136, June 2001.

[5] J. M. Bieman, S. Ghosh, and R. T. Alexander. A technique for mutation of Java objects. In *16th IEEE International Conference on Automated Software Engineering*, pages 23–26, San Diego, CA, Nov. 2001.

[6] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *SEMINAL'2001 – First International Workshop on Software Engineering using Metaheuristic INnovative ALgorithms*, Toronto, Ontario, Canada, May 2001.

[7] P. M. S. Bueno and M. Jino. Automated test data generaton for program paths using genetic algorithms. In *13th International Conference on Software Engineering & Knowledge Engineering – SEKE'2001*, pages 2–9, Buenos Aires, Argentina, June 2001.

[8] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, Sept. 1976.

[9] M. Delamaro and J. Maldonado. Interface mutation: Assessing testing quality at interprocedural level. In *19th International Conference of the Chilean Computer Science Society (SCCC'99)*, pages 78–86, Talca – Chile, Nov. 1999.

[10] M. E. Delamaro and J. C. Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS'96)*, pages 79–95, East Brunswick, NJ, July 1996.

[11] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, Mar. 2001.

[12] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation 2000 Symposium*, pages 91–101, San Jose, CA, Oct. 2000. Kluwer Academic Publishers.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, Apr. 1978.

[14] R. A. DeMillo and A. J. Offutt. Constraint based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, Sept. 1991.

[15] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[16] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis testing for finite state machines. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, pages 220–229, Monterey – CA, Nov. 1994. IEEE Computer Society Press.

[17] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis applied to validate specifications based on petri nets. In *FORTE'95 – 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols*, pages 329–337, Montreal, Canada, Oct. 1995. Kluwer Academic Publishers.

[18] A. D. Friedman. *Logical Design of Digital Systems*. Computer Science Press, 1975.

[19] S. Ghosh and A. P. Mathur. Interface mutation. *STVR – Software Testing, Verification and Reliability*, 11(4):227–247, Dec. 2001. (Special Issue: Mutation 2000 - A Symposium on Mutation Testing. Issue Edited by W. Eric Wong).

[20] M. Harman, R. Hierons, and S. Danicic. The relationship between program dependence and mutation analysis. In *Mutation 2000 Symposium*, pages 5–12, San Jose, CA, Oct. 2000. Kluwer Academic Publishers.

[21] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *STVR – Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[22] D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical Report CMU-CS-94-169, School of Computer Science – Carnegie Mellon University, Pittsburgh, PA, July 1994.

[23] B. F. Jones, D. E. Eyres, and H. H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[24] R. F. Jorge, A. M. R. Vincenzi, M. E. Delamaro, and J. C. Maldonado. Teste de mutaço: Estratégias baseadas em equivalência de mutantes para reduço do custo de aplicaço. In *CLEI'2001 – XXVII Latin-American Conference on Informatics*, Mérida – Venezuela, May 2001. (available in CD-ROM: article number – a202.pdf).

[25] M. Kim. Design of a mutation testing tool for C. Department of Computer Sciences, Purdue University, Apr. 1992.

[26] S. Kim, J. A. Clark, and J. A. Mcdermid. The rigorous generation of Java mutation operators using HAZOP. In *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, Dec. 1999.

[27] S. Kim, J. A. Clark, and J. A. Mcdermid. Class mutation: Mutation testing for object-oriented programs. In *Object-Oriented Software Systems – OOSS*, 2000. Disponvel em: `http://www.cs.york.ac.uk/~jac/`. Acesso em: 01/03/2004.

[28] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug. 1990.

[29] A. P. Mathur. Performance, effectiveness and reliability issues in software testing. In *15th Annual International Computer Software and Applications Conference*, pages 604–605, Tokio, Japan, Sept. 1991. IEEE Computer Society Press.

[30] C. Michael and G. McGraw. Opportunism and diversity in automated software test data generation. Technical Report RSTR-003-97-13, RST Corporation, Sterling, VA, Dec. 1997.

[31] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, 1997.

[32] E. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *The Journal of Software Testing, Verification and Reliability*, 9(4):205–232, Dec. 1999.

[33] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *STVR – Software Testing, Verification and Reliability*, 4:131–154, 1994.

[34] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, Jan. 1999.

[35] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, Apr. 1996.

[36] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *COMPASS'96– In Annual Conference on Computer Assurance*, pages 224–236, Gaithersburg, MD, June 1996. IEEE Computer Society Press.

[37] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *STVR – Software Testing, Verification and Reliability*, 7(3):165–192, Sept. 1997.

[38] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *STVR – Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[39] R. S. Pressman. *Software Engineering – A Practitioner's Approach*. McGraw-Hill, 5 edition, 2001.

[40] R. L. Probert and F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In *IFIP TC6 – Third International Workshop on Protocol Test Systems*, pages 57–76. North-Holland, 1991.

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

[42] D. T. Ross. Structured analysis (sa): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, Jan. 1977.

[43] A. S. Simo and J. C. Maldonado. MuDeL: A language and a system for describing and generating mutants. In *XV SBES – Simpsio Brasileiro de Engenharia de Software*, pages 240–255, Rio de Janeiro, Brasil, Oct. 2001.

[44] A. S. Simo, A. M. R. Vincenzi, J. C. Maldonado, and A. C. L. Santana. Software product instrumentation description. Technical Report 157, Instituto de Cincias Matemáticas e de Computaço – ICMC-USP, Mar. 2002.

[45] T. Sugeta. Proteum-rs/st : Uma ferramenta para apoiar a validaço de especificações statecharts baseada na análise de mutantes. Master's thesis, ICMC-USP, So Carlos, SP, Nov. 1999.

[46] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate SDL specifications. In *16th IFIP International Conference on Testing of Communicating Systems – TestCom2004*, pages 193–208, Oxford, United Kingdom, Mar. 2004. Springer.

[47] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30:61–79, 2000.

[48] R. Untch, M. J. Harrold, and J. Offutt. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–148, Cambridge, Massachusetts, June 1993.

[49] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro. Unit and integration testing strategies for C programs using mutation-based criteria. *STVR – Software Testing, Verification and Reliability*, 11(4):249–268, Dec. 2001.

[50] A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering – IJSEKE*, 12(6):675–689, Dec. 2002.

[51] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.