$\mathcal{MuDeL}$: A Language and a System for Describing and Generating Mutants

Adenilso da Silva Simão
José Carlos Maldonado
{adenilso,jcmaldon}@icmc.sc.usp.br

Departamento de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

Av. Trabalhador Saocarlense, 400
Cx. Postal 668
CEP. 13560-970
São Carlos — São Paulo

Abstract

Mutation Testing is an approach for assessing the quality of a test case suite by analyzing its ability in distinguishing the product under test from a set of alternative products, the so-called mutants. The mutants are generated from the product under test by applying a set of mutant operators, which produce products with slight syntactical differences. The mutant operators are usually based on typical errors that occur during the software development and can be related to a fault model. In this paper, we propose a language — named $\mathcal{MuDeL}$ — for describing mutant operators aiming not only at automating the mutant generation, but also at providing precision and formality to the operator descriptions. The language was designed using concepts that come from transformational and logical programming paradigms, as well as from context-free grammar theory. The language is illustrated with some examples. We also describe the `mudelgen` system, developed to support this language.

Keywords: Testing, Programming Language, Mutation Testing, Mutant Generation.

1 Introduction

Mutation Testing [3, 6] is a testing approach that has been proposed to assess the quality of a test case suite in revealing some specific classes of faults. In this sense, Mutation Testing can be classified among the fault-based testing techniques. It was originally proposed for program testing [6]. Since then, several researchers have applied its underlying concepts in a variety of other contexts, e.g. specification testing [8, 9, 20, 21], interface testing [5], protocol testing [7, 15] and network security model testing [16].

The main idea behind Mutation Testing is to employ a set of alternative products[1] (the so-called *mutants*) of the product under test (the *original* product). These mutants are derived from the original product through some syntactical changes made to induce specific faults in the product. Then, the ability of a test case suite in revealing those faults is estimated by running the mutants and comparing their results against the result of the original product in the same test cases.

The faults considered to generate the mutants are based upon knowledge about errors that typically occur during the software development and are usually related to a fault model. In the Mutation Testing approach, the fault model is partially embedded in the *mutant operators* [13]. From an abstract viewpoint, a mutant operator is a function that takes a product as input and generates a set of products in which the fault modeled by that particular operator is injected. The fault model has great impact in the Mutation Testing cost and effectiveness, and, hence, so do the mutant operators. In general, when the Mu-

[1] Throughout this paper we will use the term "product" in the same sense most testing research papers employ the term "program".

tation Testing is proposed for a particular language, one of the first steps is to describe the fault model, part of it usually in the form of a mutant operator set. The fault model, as well as the mutant operator set, has to be assessed and evolved to improve its accuracy w.r.t. to the language in question. This is usually made by theoretical and/or empirical analyses. Specifically for empirical analysis, it is necessary to design and construct a prototype or a supporting tool, once the manual generation of mutants is very costly and error-prone. However, the design and construction of this kind of tool are also costly and time-consuming tasks. An approach often used to tackle this problem is to establish prototyping mechanisms that provide a low-cost alternative, easing the experimentation with the mutant operators without requiring too much effort to be expended in implementing tools.

Another important issue to be considered is that, given the already mentioned impact in the Mutation Testing effectiveness, the mutant operators have to be precisely defined. If one takes an informal description of a mutant operator and implements it in the way one interprets it, one runs the risk of not being able to compare results with someone else, because it may not be clear whether the mutants in hand are the same. That is, mutant operators, alike any other piece of artifact of software engineering, must be described in a way as precise and formal as possible, avoiding ambiguities and inconsistencies.

In this paper we describe a language — called $\mathcal{MuDeL}$ (**MU**tant **DE**scription **L**anguage) — for the description of mutant operators. The language was designed with concepts from transformational [14] and logical [2] paradigms. Its motivation is threefold. First, we want a way to precisely and unambiguously describe the operators. In this respect, the $\mathcal{MuDeL}$ is an alternative for sharing mutant descriptions. The denotational semantics of $\mathcal{MuDeL}$ is defined in [19]. Second, the description can be "compiled" into an actual mutant operator, enabling the mutant operator designer to validate the description and potentially to improve it. With this particular purpose, we have implemented the `mudelgen` system. Given a mutant operator description and the original program, the `mudelgen` compiles the description and generates the mutants, based on a context-free grammar of the original product. And finally, by providing an abstract view of the mutations, $\mathcal{MuDeL}$ eases the reuse of mutant operators defined for different languages. For example, although the actual grammars of, say, C and Java are quite different, they both share several similar constructions, and, by carefully designing their grammars and the mutant operators, we can reuse the mutant operators that operate on the same construction, e.g., deleting statements, swapping expressions, and so on, on both languages. The $\mathcal{MuDeL}$ language captures the underlying concepts that led to the mutant operator definition.

Mutations can be classified in two major groups: context-free mutations and context-sensitive ones. Roughly speaking, context-free mutations are those that could be carried out regardless the syntactical context where the mutated part is in. Conversely, context-sensitive mutations depend upon the context, e.g. the variables visible in a specific scope. Most mutant operators in literature [1, 8, 9] involve context-free mutations. Currently, $\mathcal{MuDeL}$ only supports context-free mutations and some simple kinds of context-sensitive ones. Indeed, considering all kinds of context-sensitive mutations would require more sophisticated constructions, and, however, there are usually few cases in which those constructions would be used [1].

The Mutation Testing demands several functionalities other than just generating mutants, e.g. test cases handling, mutant execution and output checking. Both $\mathcal{MuDeL}$ and `mudelgen` are to be used as a piece in a complete mutation tool, either in a tool specifically tailored to a particular language or in a generic tool — a tool that could be used to support Mutation Testing application in (ideally) most used languages. Up to now, we have already employed $\mathcal{MuDeL}$ and `mudelgen` for describing and generating mutants for C programs and for Petri net specifications and we are currently working on mutant operator descriptions for C++ and Java programs and coloured Petri net specifications.

This paper is organized as follows. In Section 2 we present some work related to our approach. Basic concepts needed for the discussion in the remain sections are presented in Section 3. In Sections 4 and 5 we present the $\mathcal{MuDeL}$ language and the `mudelgen` system, respectively. Finally, in Section 6 we make some concluding remarks.

2 Related Work

Transformational programming paradigm is based on describing abstractly how to transform a source item in a target one through a systematic series of syntactical changes. In general, the source and the target item are in different languages. In a special case, both can be in the same language. The $\mathcal{MuDeL}$'s approach is a transformational programming, in which both the source item's and the target item's languages are the

same. In this context, the source item is the original product, whereas the target items are the mutants.

There are several examples of transformational languages and systems, e.g. DRACO [14, 18], TXL [4] and Refine [11]. With some effort, any of these could be used for generating mutants. This can be made by adapting and sometimes tricking these systems, since none of them was designed with this particular purpose in mind. For example, Kotik and Markosian [11] describe a piece of work carried out with Refine to generate test cases. The authors also suggest Refine can be used to generate mutants and present a little example of their approach. The main drawback of using these transformational systems to describe mutants is that a set of mutants are to be generated from a single original product, and most, if not all, transformational systems work with an one-to-one schema, i.e., they take *one* input product and generate *one* transformed output product. In this sense, what we believe $\mathcal{MuDeL}$ add to this scenario is a *mutant description/generation oriented language*, which brings the main idea of mutation built-in, yet being general enough to be applied to (likely and hopefully) most used product source languages.

3 Basic Concepts

In this section we present a brief introduction to grammar and language theories, needed for the discussion that follows. A thorough presentation can be found elsewhere [17]. Syntax grammars are finite devices to describe usually infinite languages. Given a grammar G, we have $L(G)$ be the set of all sentences that can be generated by the productions in G. Most, if not all, programming or computer language is characterized by a grammar. Indeed, the grammar is usually part of the sound definition of the language.

Grammars can be classified based on the kind of productions they possess. An important class is the context-free grammars. They are simple and expressive enough to catch most constructions that are usual in computer languages. Moreover, the algorithms to recognize them are computationally tractable. Context-free grammars are usually described in BNF [22]. We will refer to them as BNF grammars, as a shortcut for context-free grammar described in BNF. A BNF grammar G is formed by a four-tuple $G = (N, T, S, R)$, where N is the set of non-terminal symbols, T is the set of terminal symbols, $S \in N$ is a non-terminal symbol referred to as the initial symbol, and $R \subseteq N \times (N \cup T)^*$ is the set of production rules. Rather informally, a production rule of the form (n, α) states that the non-terminal symbol n (the *lefthand* symbol) can be replaced by the sequence α (the *righthand* symbol sequence) of terminal and non-terminal symbols without "inflicting" the grammar. Usually, the non-terminal symbols are expressed between angle brackets and the terminal symbols between single quotes. A production (n, α) is represented in the form of

$$\langle n \rangle ::= \alpha$$

When several productions have the same lefthand symbol, they can be expressed by employing a bar '|' to separate the righthands. For example, the following expression

$$\langle n \rangle ::= \alpha_1 \mid \alpha_2$$

is equivalent to

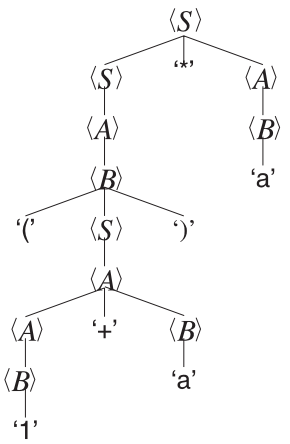
$$\langle n \rangle ::= \alpha_1$$
$$\langle n \rangle ::= \alpha_2$$

Using these conversions, BNF grammars can be (and often are) described just by means of productions, taking the terminal and non-terminal sets directly from them, and having the initial symbol be the lefthand symbol of the first production. As an example of a simple BNF grammar, Figure 1(a) presents a grammar that defines the language of parenthesis-balanced expressions formed by numbers, identifiers, additions and multiplications, preserving the usual mathematical precedence of multiplications over additions. Following the above conventions, we can derive the remaining elements of the grammar. So, we have the non-terminal set $N = \{\langle S \rangle, \langle A \rangle, \langle B \rangle, \langle \mathit{number} \rangle, \langle \mathit{identifier} \rangle\}$, the terminal set $T = \{\texttt{'*'}, \texttt{'+'}, \texttt{'('}, \texttt{')'}, \texttt{'1'}, \texttt{'a'}\}$, and the initial symbol $S = \langle S \rangle$. Observe that we actually limit ourselves to define only one production for each $\langle \mathit{number} \rangle$ and $\langle \mathit{identifier} \rangle$. Other productions could be included as well, but they would not contribute any further to the explanation that follows.

From a sequence $\gamma\langle n \rangle\delta$, we can derive another sequence of the form $\gamma\alpha\delta$, for any production $(n, \alpha) \in R$. This is represented by $\gamma\langle n \rangle\delta \Rightarrow \gamma\alpha\delta$. The language $L(G)$ defined by G is the set of all sequences of *terminal* symbols that can be derived from the *initial symbol* S with the productions in R, i.e., $\varphi \in L(G)$ if and only if $\varphi \in T^*$ and $S \Rightarrow \ldots \Rightarrow \varphi$. The derivation of φ from S can be summarized in a syntax tree for φ. The syntax tree is a tree where the non-leaf nodes are non-terminal symbols, the leaf nodes are terminal symbols and the root node is the initial symbol. If a node $\langle n \rangle$ has child nodes with labels $\alpha_1, \alpha_2, \ldots, \alpha_k$, then there has to exist a production of the form

$\langle S \rangle$ ::= $\langle A \rangle$
$\langle S \rangle$ ::= $\langle S \rangle$ '*' $\langle A \rangle$
$\langle A \rangle$ ::= $\langle B \rangle$
$\langle A \rangle$ ::= $\langle A \rangle$ '+' $\langle B \rangle$
$\langle B \rangle$ ::= $\langle number \rangle$
$\langle B \rangle$ ::= $\langle identifier \rangle$
$\langle B \rangle$ ::= '(' $\langle S \rangle$ ')'
$\langle number \rangle$ ::= '1'
$\langle identifier \rangle$ ::= 'a'

(a)



(b)

Figure 1: (a) A (partial) BNF context-free grammar of simple expressions.(b) The syntax tree for '`(1 + a) * a`'.

$\langle n \rangle := \alpha_1 \alpha_2 \ldots \alpha_k$

If traversing a syntax tree t of a grammar G and collecting the terminal symbols we obtain a sequence φ, then t is a syntax tree of φ w.r.t. G. Indeed, φ belongs to $L(G)$ if there exists such a syntax tree t for φ. Figure 1(b) presents the syntax tree for the expression '`(1 + a) * a`'.

Let π be a function that takes a grammar G and a sequence φ and returns *one* syntax tree[2] of φ w.r.t. G. Let δ be a function that takes a syntax tree and generates the sequence of terminals by traversing and collecting the leaf as alluded above. So, it holds that $\delta(\pi(G, \varphi)) = \varphi$. As stated before, a mutant operator μ is a function that takes a product and generates a set of mutated products. This can be restated by constructing a function μ' that takes a syntax tree and generates a set of mutated syntax trees, and defining μ as follows:

$$\mu(p) = \{m \mid \exists t \in \mu'(\pi(G, p)) \wedge \delta(t) = m\}$$

where p is the original product. $\mathcal{MuDeL}$ can be thought of as a language for describing μ' functions, i.e., for describing how the syntax tree of the original product is to be translated into the syntax trees of the mutants. Nevertheless, $\mathcal{MuDeL}$ also embodies both π and δ, the parsing and unparsing functions, respectively.

[2]Actually, for some grammars there can exist more than one syntax tree for a given sequence. For this case, π could choose anyone of them, by chance. To make this choice unique, we have π choose the leftmost derivation [10].

3.1 Context-Sensitiveness and Mutation

The syntax of most languages is defined by means of context-free grammars, e.g. the programming languages C and Pascal. However, these languages are, indeed, not context-free. For instance, an expression "`x = 1`" will or not be accepted as a valid C expression depending upon the context (particularly, the variable declaration context) it appears in. In this respect, the C language is clearly context-sensitive. If so, why is it usually described as a context-free grammar? The most important reasons for this are: (a) context-sensitive grammars are harder to specify and more expensive to recognize; (b) the context-sensitiveness can be properly tackled using *lookup tables*.

In a very similar way, mutations can be divided up into context-free and context-sensitive classes. Even for a context-sensitive language, there can be context-free mutations. An example of context-free mutations is the change of "`x = 1`" into "`x += 1`", since wherever the first expression is valid, so is the second. However, the change of "`x = 1`" into "`y = 1`" is context-sensitive, since the second expression will not be valid unless `y` has the same declaration status as does `x`. To deal with this rather difficult problem, a language for describing mutants should either embody features to specify context-sensitive grammar or provide some way to gather information from the context in some kind of lookup table.

Although some kind of context-sensitive mutations can be described in $\mathcal{MuDeL}$, its primarily intent is to describe context-free mutations. As pointed out before, context-sensitive grammars are harder to couple with, and so are context-sensitive mutations. More-

over, in experiments we have conducted with the $\mathcal{MuDeL}$ and `mudelgen` in generating mutants for Petri nets specifications and C programs we observed that only a small percentage of the overall existing mutant operators could not be described with context-free mutations. For example, we were able to describe all of the 11 mutant operators proposed by Fabbri [7] for Petri net specifications. For C programs, we were able to specify 59 out of 71 mutant operators proposed by Agrawal [1]. Currently, we are investigating mechanisms for tackling this problem and we will extend $\mathcal{MuDeL}$'s ideas to cope with context-sensitive in mutant operators.

4 $\mathcal{MuDeL}$: The Language

We introduce a set $\mathcal{M}$ of *meta-variables* and extend the syntax tree by allowing for leaves to be meta-variables as well as terminal symbols. Moreover, in this extension, the root node can be any non-terminal symbol (not only the initial one, as in syntax trees). We call these extended syntax trees *pattern trees*, or, if it is unambiguous from the context, just *patterns*. Each meta-variable has an associated non-terminal symbol, which is called its *type*. A meta-variable can be either free or bound. Every bound meta-variable is associated to a sub-tree that can be generated from its type. Therefore, a syntax tree is just a special kind of pattern tree; a kind where every meta-variable (if any) is bound. Figure 2 shows an example of a pattern tree. As a way to distinguish from ordinary identifiers, we prefix the meta-variables with a colon (:). Even in the presence of meta-variables, the children of a node must be in accordance with its productions, i.e., a meta-variable can only occur where a non-terminal of its type also could.

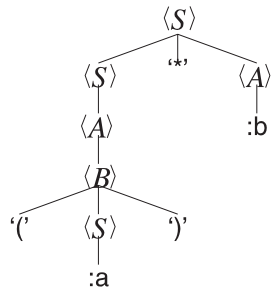


Figure 2: The pattern tree for ‘`( :a ) * :b`’. The types of ‘`:a`’ and ‘`:b`’ have been declared as $\langle A \rangle$ and $\langle B \rangle$, respectively.

To specify patterns we use the following notation. The simplest pattern is formed by an anonymous meta-variable, as its root node. This pattern is expressed just by the non-terminal symbol that is its root node enclosed in squared brackets. For example, [A] is a pattern whose root node is an anonymous meta-variable of type $\langle A \rangle$. Most times, such a simple pattern will not be enough to specify pattern trees. For those situations, one can use a more elaborated pattern denotation. The non-terminal root symbol is put in squared brackets, as before, but following it, in angle brackets, is included a sequence of terminal symbols and meta-variables that should be parsed to generated the pattern tree. For example, the pattern tree in Figure 2 is denoted by [S< (:a) * :b>]. Note that inside the angle brackets the grammar of the product, rather than the $\mathcal{MuDeL}$'s grammar, is to be respected. Nonetheless, meta-variables come from $\mathcal{MuDeL}$ itself and, thus, the previous pattern will only be valid if the meta-variables :a and :b are declared with proper types.

Indeed, the best term for $\mathcal{MuDeL}$, instead of a language, is a *meta-language*, in that a $\mathcal{MuDeL}$'s description is valid or not w.r.t. a given source grammar. That is, given a source grammar of a product language, we instantiate a $\mathcal{MuDeL}$ language for that grammar. The source grammar, for example, determines the form and the syntax of the pattern trees.

4.1 Matching and Replacing

There are two main operations in the $\mathcal{MuDeL}$ language: *matching* and *replacing*. For matching, we take two pattern trees c and m and try to unify them, using the same algorithm as the Prolog language [2]. A matching can either fail or succeed. In case of success, the meta-variables in the pattern tree are unified either to closed pattern trees or to other meta-variables, in a way that makes them unrestrictly interchangeable. In case of failure, no meta-variable unification occurs. The unification can bind some meta-variables either to other meta-variables or to sub-trees. Binding a meta-variable to another meta-variable leads to situations where a chain of meta-variable referencings is obtained. In this case, the actual value referred to by a meta-variable is the value of the last meta-variable in a chain, and it can be either a sub-tree or be free. If a pattern tree is such that the referents of all its meta-variables are sub-trees, it is called a *closed* pattern tree. The matching process is illustrated in Figure 3.

For replacing, we take three patterns c, r and b, try to unify c with r and, in case of success, substitute c by b. This is actually the most general operation, in that the matching is just a special case where b equals

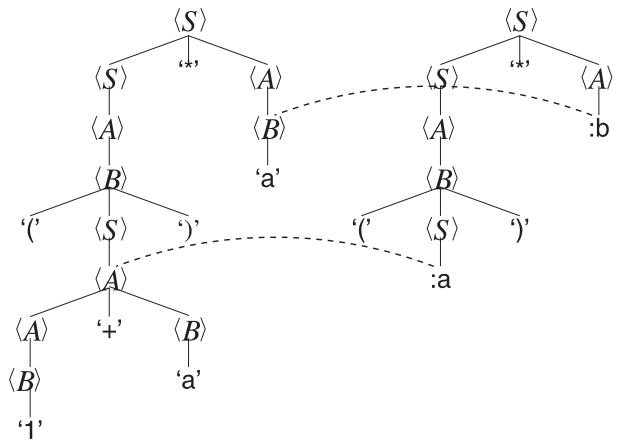


Figure 3: Matching Process

r, i.e., no change is made at all. A replacement, when occurs, takes place after the unification of c and r. Thus, meta-variables can be used to make the pattern b refer to some parts of c and/or r.

The matching and replacing processes are usually employed techniques. Indeed, most of the transformational systems use a similar approach. Maybe the only novelty of $\mathcal{MuDeL}$ is the *later* unification, in a way similar to what happens in Prolog. The replacing pattern need not be closed, in that it is allowed for free meta-variables to appear. Latter on in the computation, if the same meta-variable is unified to a pattern, previous patterns in which that meta-variable has appeared will unify accordingly. This particular feature was introduced to couple with situations where the information to complete a mutation appears only latter in the product. For example, when the actual mutated value of an expression will only be available afterwards, when, say, the return value of the function is known.

4.2 $\mathcal{MuDeL}$'s Constructions

In order to generate the mutants, the operations of replacing and matching discussed in Section 3 have to be expressed in a concrete syntax. Moreover, there should exist some form to combine simple operations in more complex ones. In the remaining of this section we introduce the $\mathcal{MuDeL}$'s syntax for both expressing operations and combining them by means of some exemplifying mutant operators for C language. (Part of) the context-free grammar used in these operators (shown in Figures 4 through 11) is presented in Section 5.

A mutant operator description has three main sections. The operator name declaration comes in the first one. This name is just for documentation purposes and has no impact in the remaining declarations. Next, there is the optional section of meta-variable declarations. If present, this section is started by the keyword **var** followed by a list of one or more meta-variable declarations. A meta-variable declaration is a meta-variable name followed by a pattern tree. The last section, enclosed by the keywords **begin** and **end operator**, is a (possibly compound) operation. This operation will be executed on the syntax tree of the original product and will generate the mutants, if any. Figure 4 presents a mutant operator description, illustrating its overall structure. This mutant operator, whose name is STDL, declares the meta-variable :s with the type ⟨*statement*⟩, and has a simple operation, that, as will be clarified later, generates mutants replacing nodes with type ⟨*statement*⟩ by a semi-colon (the null statement), according to the grammar of the C programming language.

```
operator STDL
   var :s [statement]
begin
   * replace [statement<:s>]
     by      [statement<;>]
end operator
```

Figure 4: A Simple Mutant Operator. For every statement in the program, a mutant is generated by "deleting" the statement.

The replacing operation is encoded with the construction **replace** r **by** b, where r is the pattern to be found and b is the replacing pattern. This kind of construction has an implicit context pattern against which r is to be matched. By default, the context pattern of an operation is the innermost declared context pattern, or the complete product tree, if none is declared. Actually, the replacing occurs by exchanging the context pattern by the b pattern. Figure 5 shows the mutant operator _SWDD that illustrates the replacing operation. This operator changes a C *while* statement into a C *do-while* one.

The _SWDD does not really do what we would expect for such a kind of mutant operator. Recall that the implicit context defaults to the whole product tree, in this case, the whole C program. So, what _SWDD really does is to generate *one* mutant provided the whole program is just a single *while* statement (which is not even a valid C program). What would be expected is that all *while* statement, in any level, be exchanged. To accomplish this effect, we can

```
operator _SWDD
   var :e [expression] :s [statement]
begin
   replace [statement<while(:e) :s>]
   by      [statement<do :s while(:e);>]
end operator
```

Figure 5: Replace a *while* statement by a *do-while* statement.
See also Figure 6.

add the modifier "∗" (read *in depth*) to the replacing operation. The modifier "∗" means, informally, that the modified operation will be applied to the context pattern tree and recursively to any subtree thereof. Therefore, an operation with the in depth modifier will usually generate a set of mutants, since whenever one of its subtrees matches the pattern a different mutant is produced. Figure 6 shows a somewhat more suitable description of the SWDD.

```
operator SWDD
   var :e [expression] :s [statement]
begin
  * replace [statement<while(:e):s>]
    by      [statement<do :s while(:e); >]
end operator
```

Figure 6: Replaces every *while* statement in any level by the respective *do-while* statement.

Suppose now we are designing a mutant operator that exchanges not only *while* by *do-while*, but also the other way round[3]. Changing a *do-while* statement into a *while* can be easily make by swapping the patterns in SWDD. A further construction is needed to compose both features together. For that purpose, $\mathcal{MuDeL}$ has the composition operator "||" (read *choice*). The *choice* operator takes a sequence of operations and means that each will be tried in order to generate mutants. More than one operation can be included, separating each pair with a "||", e.g. a||b||c. All operations in a chain separated by the choice composition operator is called a *choice list*. For each operation, the global state will be the same, i.e., after one operation has been concluded, the state is restored and any eventual change is undone. In other words, in any mutant, only the effects of one of these operations will be present. Therefore, the set of mutants generated by a choice list is the union of the mutants generated by each individual operation. Being able to compose different operations in this way is useful in situations in which the mutant operator requires changes in syntactically different elements of the product language, e.g. changing the expression that controls an iteration loop statement in C language, i.e. in a *for*, a *while* or a *do-while* statement, as is made by the SMTC operator proposed by Agrawal [1]. Figure 7 shows a description of the mutant operator for exchanging *while* and *do-while* by each other.

[3]Usually, this goal is accomplished by design two mutant operators, as in [1]. We made both together just for sake of illustration.

```
operator SWDDW
   var :e [expression] :s [statement]
begin
  * replace [statement<while(:e):s>]
    by      [statement<do :s while(:e);>]
  ||
  * replace [statement<do :s while(:e);>]
    by      [statement<while(:e) :s>]
end operator
```

Figure 7: Replace every *while* statement in any level by the respective *do-while* statement and vice-versa.

One more thing has to be said about the *choice* composition operator. There exists a primitive operation named **cut**, which, when executed in any operation, makes all remaining operations in the innermost choice list be skipped. The **cut** is similar to, and was design after, the Prolog's cut predicate [2]. It is primarily intended to improve control over the number of times a set of operations is executed.

Sometimes it may be necessary to execute more than one operation in the same mutant. Usually, this happens to constraint further the pattern. It can be the case, as well, that the mutant operator semantics requires more than one change be done. The composition operator ";;" (read *then*) was designed to tackle this point. The *then* operator takes a series of operations and means that each one has to be executed, in sequence, to generate a mutant. More than one operation can be included separating each pair with a ";;", e.g. a;;b;;c. The ";;" operator has a higher precedence than has "||", so a;;b||c;;d is interpreted as a choice list composed of two (compound) operations, namely, a;;b and c;;d. Double parenthesis can be used to group the operations in order to override the precedence, as in a;;((b||c));;d. However, unlike most programming language, ((a;;b))||c is not the same as a;;b||c, in that, for the latter, if either a or b is the **cut** operation, c will be skipped, whereas for the former, c will be executed anyway. In other words, the grouping chops the scope of the **cut** operation.

The matching operation is coded with the construction **match** m. As in the replacing operation, m is matched against the context pattern. If m matches the context pattern, the generation continues further, keeping the effects of any unification that ever occurs. However, if it does not match, the generation is stopped and an alternative is tried (either due to a choice composition operator or to an *in depth* modifier).

Figure 8 illustrates both *then* compositions and matching operations. It presents a mutant operator that changes the control expression of every *do-while* statement into 0, unless this expression already syntactically equals 0.

```
operator EDWT
   var :e [expression] :s [statement]
begin
  * replace [statement<do :s while(:e);>]
    by      [statement<do :s while(0);>]
  ;;
  ## :e does not match the expression 0
  :e @@ ~ match [expression< 0 >]
end operator
```

Figure 8: Enforce that the *do-while* body statement be executed at most once. The character # marks a comment line.

In operator EDWT there are also two new modifiers, the "@@" (read *apply*) and "~" (read *not*). The *apply* modifier is used to explicitly declare the context pattern. The construction :v @@ op demands the operation op be executed using the meta-variable :v as its context pattern. The *not* modifier is intended to negate the operation it is used in. The construction ~ op fails if the operation op succeeds, and succeeds with no unification if op fails.

There are also two simple operations in $\mathcal{MuDeL}$, which do not actually involve matching or replacing: the **donothing** and the **abort**. The **donothing** operation actually does not change the tree and is primarily intended to be used as a syntax placeholder where an operation is demanded. So, the simplest mutant operator that can be described in $\mathcal{MuDeL}$ is presented in Figure 9. Indeed, this operator generates only one mutant that actually equals to the original product.

In contrary to **donothing**, there is the operation **abort**, that does not allow the mutation to occur. That is, whenever an **abort** is executed, no mutant is generated. It is primarily intended as a guard to, for example, inhibit equivalent or undesirable mutants be yielded. For example, the mutant operator in Figure 10 does not generate any mutant at all.

```
operator DONOT
begin
   donothing
end operator
```

Figure 9: Generates one "mutant", yet equals to the original product.

```
operator ABRT
begin
   abort
end operator
```

Figure 10: Does not generate any mutant.

By their own, both **donothing** and **abort** operations are not really useful. However, in conjunction with matching and replacing, they are very useful to increase the control over the mutations.

4.3 An Example

In this section we present an example to better illustrate the main concepts of the $\mathcal{MuDeL}$ language. In Figure 11 we present the SDWE operator that is meant to change every *while* statement into a *do-while* and also change the control expression into 0 and 1, if it does not already equal to 0 and 1, respectively. Figure 12(a) presents a simple C program and Figures 12(b)-(e) present the mutants that will be generated for this program with the SDWE operator.

```
1 operator SDWE
2    var :e [expression] :s [statement]
3 begin
4   * replace [statement<while(:e) :s>]
5     by      [statement<do :s while(:e);>]
6   ;;
7   :e @@ ((
8       ~ match [expression< 0  >]
9       ;;
10      replace [expression]
11      by      [expression< 0  >]
12  ||
13      ~ match [expression< 1  >]
14      ;;
15      replace [expression]
16      by      [expression< 1  >]
17  ||
18      donothing
19  ))
20 end operator
```

Figure 11: A multi-purpose *while* mutant operator.

```
int main() {
   while( 1 ) {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      while ( c > 'A' ) c--;
   }
}
```
(a) Original Program

```
int main() {
   do {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      while( c > 'A' ) c --;
   } while( 0 );
}
```
(b) Mutant #1

```
int main() {
   do {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      while ( c > 'A' ) c --;
   } while( 1 );
}
```
(c) Mutant #2

```
int main() {
   while( 1 ) {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      do c --; while( 0 );
   }
}
```
(d) Mutant #3

```
int main() {
   while( 1 ) {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      do c --; while( 1 );
   }
}
```
(e) Mutant #4

```
int main() {
   while( 1 ) {
      int c;
      if( ( c = getchar() ) == EOF )
         break;
      do c --; while( c > 'A' );
   }
}
```
(f) Mutant #5

Figure 12: (a) Original Program. (b)-(e) Mutants generated by operator in Figure 11.The mutated parts of the code are highlighted.

The replacing operation in lines 4 and 5 changes every *while* statement into a *do-while* statement, in any depth. The meta-variable :e stands for the control expression of the *while*. The group of operations in lines 7 through 19 make changes in this control expression. Observe that the context pattern declaration in line 7 affects the whole group, and, consequently, every operation therein.

The (negated) matching in line 8 makes sure that the context pattern (:e, in this case) is not equal to 0. If so, the context pattern is changed to 0, by the replacing in lines 10 and 11, and a mutant is generated. Note that these two operations compose a sequence, which is part of a choice list. Then, the next choice is tried, in this turn w.r.t. the expression 1. Finally, the **donothing** operation in line 18 is tried and a mutant is generated only with the replacement of line 4.

Analyzing how the mutants are generated in this example illustrates the way $\mathcal{MuDeL}$ processes a mutant operator description. The replacing operation (lines 4 and 5) is marked with the *in depth* modifier and, therefore, the whole program syntax tree will scanned, looking for nodes that match the respective pattern and changing them accordingly. The replacing operation and the group of operations in lines 7 through 19 compose a sequence, i.e. every mutant should include the effects of the replacing *and* the effects (if any) of the group. This group, by its turn, is composed by a list of three choices: the first choice is in lines 8 through 11, the second one is in lines 13 through 16, and the last one is in line 18. Only the effects (if any) of one of these choices will be included in a particular mutant. For instance, Mutants #1 and #2 in Figures 12(b)-(c) are generated by replacing the outermost while of the program in Figure 12(a) and applying the first and the third choices, respectively. (Observe that the second

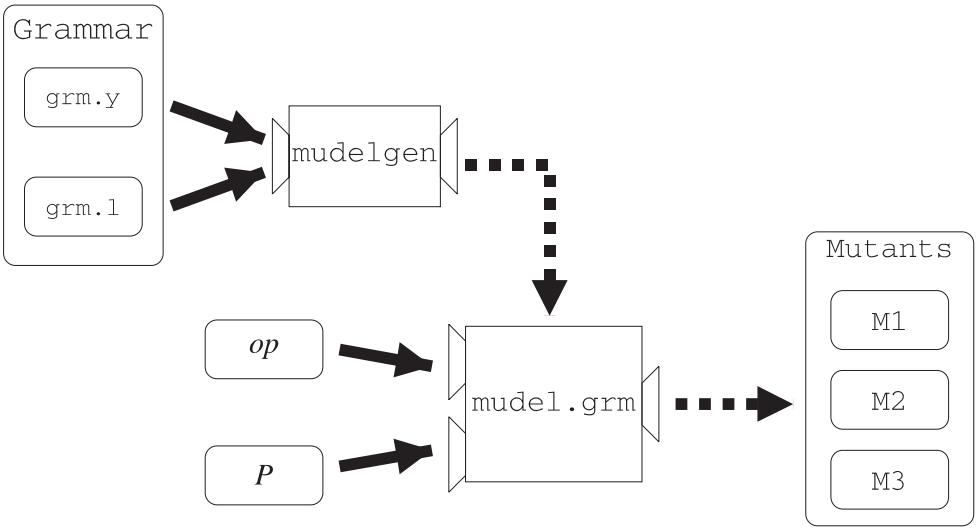


Figure 13: `mudelgen` Execution Schema

choice does not generate a mutant, since the operation in line 13 does not succeed.) On the other hand, Mutants #3, #4 and #5 in Figures 12(d)-(f) are generated by replacing the innermost while and applying each of the choices, respectively.

5 `mudelgen`: The System

In addition to being able to precisely describe the mutant operators, it is desirable to be able to generate the mutants themselves. So, given a grammar, given a product in that grammar, and given a mutant operator for that grammar, we should generate the respective mutants.

For this task, we developed the `mudelgen` (standing for $\mathcal{M}u\mathcal{D}e\mathcal{L}$ **Gen**erator). When `mudelgen` is input with a grammar, say `grm`, in a special format described latter, it produces a program called `mudel.grm`. In its turn, this program is to be run with a mutant operator description *op* and a product P. After checking whether both *op* and P are syntactically correct w.r.t. the input grammar, a mutant set M is generated. The overall execution schema of `mudelgen` is stretched in Figure 13.

The grammar input to `mudelgen` is furnished in two files: the `.y` and the `.l`. The `.y` file is the context-free grammar, written in a subset of YACC syntax [12]. The terminal symbols should be explicitly declared with the construction `%token` and usually come in uppercase. A production of the form

$$\langle n \rangle ::= \text{A b} \mid \text{c}$$

is encoded as

```
n : A b | c ;
```

The `.l` file is a lexical analyzer and gives the actual form of the terminal symbols of the grammar. It is encoded in a subset of the LEX syntax [12]. As an example of the $\mathcal{M}u\mathcal{D}e\mathcal{L}$'s input files, Figure 14 presents part of the `C.l` and `C.y` files, the lexical analyzer and the grammar, respectively, for the C language. Indeed, these files can be thought of as minimal standard `yacc` and `lex` inputs, from which all so-called semantic actions were stripped off.

An example of a typical execution with `mudelgen` is presented following.

```
hortencia% mudelgen C
hortencia% mudel.C SDWE.op Sample.C
```

In the first line, `mudelgen` is called with the option `C`, what means that files named `C.l` and `C.y` are to be read. Then, the program `mudel.C` is produced. In second line, this program is run with the files `SDWE.op` and `Sample.C`, which are the mutant operator description and the original product, respectively. As result, a set of mutants is generated, with each single mutant in an own file.

To implement the `mudelgen`, an alternative approach could have been taken, instead of the one alluded in Figure 13, by having `mudelgen` take the mutant operator description as an input and, so, generating a program specific for the operator, say `mudel.op.grm`. This program could be somehow optimized for that specific mutant operator. However,

```
/* C.y file */
%token WHILE
%token DO
%token OPENPAR
%token CLOSEPAR
%token SEMICOLON
%token CONSTANT
...
%%
statement
   : iteration_statement
   | null_statement
   | ...  ;

iteration_statement
   : WHILE OPENPAR expression CLOSEPAR
       statement
   | DO
       statement
     WHILE OPENPAR expression CLOSEPAR
     SEMICOLON
   | ...
        ;

null_statement : SEMICOLON ;

expression : CONSTANT | ...  ;
```

(a)

```
/* C.l file */
...
"do"     { RETURN(DO); }
"while"  { RETURN(WHILE); }
"("      { RETURN(OPENPAR); }
")"      { RETURN(CLOSEPAR); }
";"      { RETURN(SEMICOLON); }
[0-9]+   { RETURN(CONSTANT); }
...
```

(b)

Figure 14: (a) Part of the `C.y` input file. (b) Part of the `C.l` input file.

we have chosen the approach in Figure 13 because we believe that in the early phases of a mutant operator description, much experimentation would be carried out, but more with the mutant operator, and lesser with the grammar. So, we tried to isolated the potential bottleneck, avoiding having to run `mudelgen` too many times, a rather longer process.

We have also implemented a prototyping graphical interface — called $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator — for easing the visualization of the execution of a mutant operator. Currently, $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator has only some limited features that allows to inspect the log of execution, without being able to interfere in the process. Figure 15 presents the main window of $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator. At the top of the window are the buttons that control the execution of the animator, such as *Step*, *Exit* etc. The remaining of the window is divided up into four areas:

$\mathcal{M}u\mathcal{D}e\mathcal{L}$ Description: In the bottom left area, $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator presents the mutant operator description. A rectangle indicates which line is currently executing. Every meta-variable is highlighted with a specific color. The same color is used in whichever occurrence of the same meta-variable throughout all the other areas.

Mutant Tree: In the top left area, the animator shows the syntax tree of the product, reflecting any change so far accomplished by the execution. An arrow indicates which node is currently the context tree. Whichever meta-variable binding is presented by including the name of the meta-variable above the respective tree node.

Pattern Tree: In the top right area, $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator shows the tree of the pattern currently active (i.e., in the current line) in the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Description area.

Current Product: In the bottom right area, the current state of the product, obtained by traversing the current state of the mutant tree, is presented. The parts in the mutant that correspond to the nodes bound to meta-variables are highlighted with the respective color.

Since $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Animator enables us to observe the execution of a mutant operator description, it is very useful not only for obtaining a better understanding of the $\mathcal{M}u\mathcal{D}e\mathcal{L}$'s mechanisms, but also for (passively) debugging a mutant operator.

6 Concluding Remarks

The efficacy of Mutation Testing is heavily related to the quality of the mutants employed. Mutant operators, therefore, play a fundamental role in this scenario, since they are used to generate the mutants. Due to their importance, mutant operators should be precisely defined. Moreover, they should be experimented with and improved. However, implementing tools to support experimentation is very costly and time-consuming.

In this paper we presented the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language as a device for describing mutant operators. The language is based on the transformational paradigm and also uses some concepts from logical programming. Being described in $\mathcal{M}u\mathcal{D}e\mathcal{L}$, an operator can be "compiled" and the respective mutants can be generated using the `mudelgen` system. $\mathcal{M}u\mathcal{D}e\mathcal{L}$ and `mudelgen` together form a powerful instrument in developing and validating mutant operators.

$\mathcal{M}u\mathcal{D}e\mathcal{L}$ was mainly design to deal with context-free mutations. With this decision, we keep the language quite simple, yet considerably expressive. However, there are some important kinds of mutants that are inherently context-sensitive and can not be described in $\mathcal{M}u\mathcal{D}e\mathcal{L}$. For example, some program mutant operators might need knowledge about the variables defined prior some specific point in the program. Currently, we are investigating how to cope with this setback.

$\mathcal{M}u\mathcal{D}e\mathcal{L}$ language has some other constructions that were not discussed in this paper. For example, there are constructions for declaring and invoking rules. Rules can be thought of as procedures of conventional programming languages. The rule declarations can be recursive, i.e., a rule can invoke itself. This feature does enlarge the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language expressiveness by allowing for an alteration to be repeatedly applied in the same mutant. Another characteristic not discussed in this paper is the built-in rules. Their are provided to cope with tasks that are hard, cumbersome or even impossible to be carried out only with the construction $\mathcal{M}u\mathcal{D}e\mathcal{L}$ embodies, e.g. arithmetics, string manipulation, and so on. We, then, keep the kernel of $\mathcal{M}u\mathcal{D}e\mathcal{L}$ tiny, whereas we provide built-in rules for any further need we have to take care of. We are currently developing an API (Application Programming Interface) to allow the implementation and inclusion of rules written in a conventional programming language, namely, in C^{++}.

Our forthcoming steps in this research include:

- to develop an integrated development environment (IDE), providing features to edit the

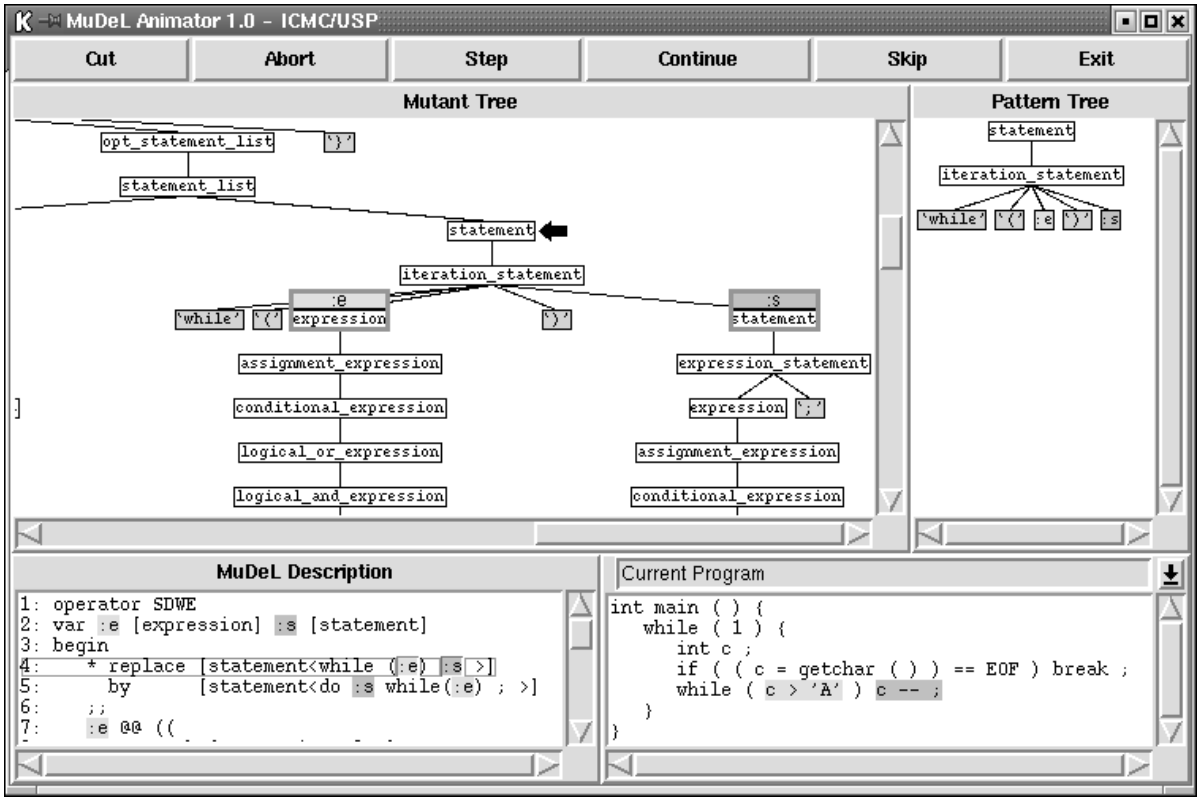


Figure 15: MuDeL Animator.

context-free grammar, the mutant operator and the original product in an as uniform manner as possible, as well as providing features to compile, execute and debug the mutant operator description. `mudelgen` is currently operated by means of command-line invocations and has some limited graphical interaction (through $\mathcal{MuDeL}$ Animator). To ease the usage and experimentation, an IDE would be more appropriate.

- to further investigate the context-sensitiveness of some kinds of mutants and devise constructions to cope with them. As already stated, $\mathcal{MuDeL}$ does not currently address problems concerning context-sensitive mutations.
- to integrate the $\mathcal{MuDeL}$ and the `mudelgen` in a complete mutation tool. Mutation Testing demands also other activities such as test case handling, mutation execution, result analysis, and so on. We are now specifying and designing a complete mutation tool which follows the main ideas of $\mathcal{MuDeL}$, i.e., a tool with multi-language support.

Acknowledgements

The authors would like to thank the Brazilian Funding Agencies — CNPq, FAPESP and CAPES — and the Telcordia Technologies (USA) for their partial support to this research. The authors would also like to thank the anonymous referees for their valuable comments.

References

[1] Agrawal, H. (1989). Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center/Purdue University.

[2] Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, England ; Reading, Mass., 2 edition.

[3] Budd, A. T. (1981). *Mutation Analysis: Ideas, Examples, Problems and Prospects*, pages 129–148. Computer Program Testing. North-Holland Publishing Company.

[4] Cordy, J. R. and Shukla, M. (1992). Practical metaprogramming. In *Centre for Advanced Studies Conference*, pages 215–224, Toronto.

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

[6] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

[7] Fabbri, S. C. P. F. (1996). *A Análise de Mutantes no Contexto de Sistemas Reativos: Uma Contribuição para o Estabelecimento de Estratégias de Teste e Validação*. PhD thesis, IFSC/USP, São Carlos, SP.

[8] Fabbri, S. C. P. F., Maldonado, J. C., Delamaro, M. E., and Masiero, P. C. (1999a). Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In *XIX SCCC - International Conference of the Chilean Computer Science Society*, pages 96–104, Talca, Chile.

[9] Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., and Masiero, P. C. (1999b). Mutation testing applied to validate specifications based on statecharts. In *ISSRE — International Symposium on Software Reliability Systems*, pages 210–219.

[10] Fejer, P. A. and Simovici, D. A. (1990). *Mathematical Foundations of Computer Science*, volume I. Springer-Verlag.

[11] Kotik, G. B. and Markosian, L. Z. (1989). Automating software analysis and testing using a program transformation system. In *Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification*, pages 75–84.

[12] Mason, T. and Brown, D. (1990). *Lex & Yacc*. O'Reilly.

[13] Nakagawa, E. Y. and Maldonado, J. C. (2001). Software-fault injection based on mutant operators. In *Anais do XI Simpósio Brasileiro de Tolerância a Falhas*, pages 85–98, Florianópolis, SC.

[14] Neighbors, J. (1984). The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574.

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

[16] Ritchey, R. W. (2000). Mutating network models to generate network security test cases. In *Mutation 2000*, pages 101–108, San Jose, California.

[17] Salomaa, A. (1973). *Formal Languages*. Academic Press, New York.

[18] Santana, A. C. L., Prado, A. F., and Lopes de Souza, W. (1997). Utilização do paradigma Draco para implementar especificações Estelle na linguagem C++. In *Anais do 15o Simpósio Brasileiro de Redes de Computadores*, pages 118–134, São Carlos, SP.

[19] Simão, A. S., Maldonado, J. C., and Bigonha, R. S. (2002). Using denotational semantics in the validation of the compiler for a mutation-oriented language. In *Proceedings of V Workshop of Formal Methods*, pages 4–19, Gramado, RS.

[20] Simão, A. S., Maldonado, J. C., and Fabbri, S. C. P. F. (2000). Proteum-RS/PN: A tool to support edition, simulation and validation of Petri nets based on mutation testing. In *Anais do XIV Simpósio Brasileiro de Engenharia de Software*, pages 227–242, João Pessoa, PB.

[21] Souza, S. R. S., Maldonado, J. C., Fabbri, S. C. P. F., and Lopes de Souza, W. (2000). Mutation testing applied to Estelle specifications. *Quality Software Journal*, 8(4):285–301. Also published in 33^{rd} Hawaii Internacional Conference on System Sciences, 2000.

[22] Vladimir, D. (1989). *Formal Languages and Automata Theory*. Computer Science Press.