Computer Languages, Systems & Structures 35 (2009) 322 – 339

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A transformational language for mutant description

Adenilso Simão[a,*], José Carlos Maldonado[a], Roberto da Silva Bigonha[b]

[a]Departamento de Sistemas de Computação, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Av. Trabalhador São-carlense, 400 - Centro, P.O. Box: 668, ZIP: 13560-970 São Carlos, SP, Brazil
[b]Departamento de Computação, Universidade Federal de Minas Gerais, Brazil

ARTICLE INFO

Article history:
Received 9 December 2006
Accepted 29 October 2008

Keywords:
Mutation testing
Transformation languages
Logical languages
Software testing
Prototyping

ABSTRACT

Mutation testing has been used to assess the quality of test case suites by analyzing the ability in distinguishing the artifact under testing from a set of alternative artifacts, the so-called mutants. The mutants are generated from the artifact under testing by applying a set of mutant operators, which produce artifacts with simple 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 $\mathscr{MuDeL}$ (MUtant DEfinition Language)—for the definition of mutant operators, aiming not only at automating the mutant generation, but also at providing precision and formality to the operator definition. The proposed language is based on concepts from transformational and logical programming paradigms, as well as from context-free grammar theory. Denotational semantics formal framework is employed to define the semantics of the $\mathscr{MuDeL}$ language. We also describe a system—named `mudelgen`—developed to support the use of this language. An executable representation of the denotational semantics of the language is used to check the correctness of the implementation of `mudelgen`. At the very end, a mutant generator module is produced, which can be incorporated into a specific mutant tool/environment.

1. Introduction

Originally, mutation testing [1,2] is a testing approach to assess the quality of a test case suite in revealing some specific classes of faults, and can be classified as a fault-based testing technique. Although it was originally proposed for program testing [2], several researchers have applied its underlying concepts in a variety of other contexts, testing different kinds of artifacts, e.g., specifications [3–7], protocols testing [8] and network security models [9]. Moreover, mutation testing has been employed as a useful mechanism to improve statistical validity when testing criteria are compared, such as in [10].

The main idea behind mutation testing is to use a set of alternative artifacts (the so-called *mutants*) of the artifact under testing (the *original* artifact) to evaluate test case sets. These mutants are generated from the original artifact by introducing simple syntactical changes and, thus, inducing specific faults. Usually, only a simple modification is made in the original artifact. The resulting mutants are the so-called 1-*order* mutants [11]. A *k-order* mutant can be thought of as a mutant in which several 1-*order* mutations were applied [12]. The ability of a test case suite in revealing these faults is checked by running the mutants and comparing their results against the result of the original artifact for the same test cases.

The faults considered to generate the mutants are based upon knowledge about errors that typically occur during software development and can be associated to a fault model. In the mutation testing approach, the fault model is embedded in the *mutant operators* [13]. A mutant operator can be thought of as a function that takes an artifact as input and produces a set of mutants,

* Corresponding author. Tel.: +55 16 3373 9700; fax: +55 16 3371 2238.
E-mail addresses: adenilso@icmc.usp.br (A. Simão), bigonha@dc.ufmg.br (R. da Silva Bigonha).

1477-8424/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2008.10.001

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 does the mutant operator set. In general, when the mutation testing is proposed for a particular artifact, one of the first steps is to describe the fault model and a mutant operator set.

Considering the important role of mutant operators to the mutation testing, their definition and implementation are basic issues for its efficient and effective application. The mutant operator set has to be assessed and evolved to improve its accuracy w.r.t. the language in question. This is usually made by theoretical and/or empirical analysis. Specifically for empirical analysis, it is necessary to design and construct a prototype or a supporting tool, because manual mutant generation is very costly and error-prone. However, the tool design and construction are also costly and time-consuming tasks. An approach that can be used to tackle this problem is to establish prototyping mechanisms that provide a low-cost alternative, making easier the evaluation and evolution of the mutant operators without requiring too much effort to be expended in developing tools. At the very end, the produced mutant generator module may be incorporated into a specific mutant tool/environment.

Another important issue to be considered is that, given the already mentioned impact on the mutation testing effectiveness, mutant operators must be described in a way as rigorous as possible, in order to avoid ambiguities and inconsistencies. This is similar to what happens to other artifacts of software engineering. Several initiatives towards defining mutant operators for different programming languages can be found in the literature [14–19]. Although we can identify some approaches in which the operators are formally defined (e.g., [20]), in most of the cases, the definition is informal and based on a textual description of the changes that are required in order to generate the mutants (see, e.g., [14]).

From these works, we can observe that there are common conceptual mutations amongst different languages, such as Fortran, C, C++, Statecharts, FSMs and so on, although this point has not been explicitly explored by the authors. This fact motivated us to investigate mechanisms to design and validate mutation operators as independent as possible of the target language. This same scenario leads to opportunities to reuse the knowledge underlying the mutations (i.e., effectiveness, costs related to the generation of mutants, to determination of equivalent mutants, to the number of test cases required to obtain an adequate test case set) of particular mutations, and of the related operators [21,22].

In this paper, we present a language—called $\mathscr{MuDeL}$ (MUtant DEfinition Language)—for the definition of mutant operators, a tool to support the language and case studies that show how these mechanisms have been employed in several different contexts. The language was designed with concepts from transformational [23] and logical [24] paradigms. Its motivation is threefold. Firstly, $\mathscr{MuDeL}$ provides a way to precisely and unambiguously describe the operators. In this respect, $\mathscr{MuDeL}$ is an alternative for sharing mutant descriptions. We employed *denotational semantics* [25,26] to formally define the semantics of $\mathscr{MuDeL}$ language [27]. Observe that the description of the mutant are syntax driven and the semantics of the mutant itself are not taken into consideration. Secondly, the mutant operator 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 purpose, we have implemented the `mudelgen` system. Given a mutant operator defined in $\mathscr{MuDeL}$ and the original artifact, the `mudelgen` compiles the definition and generates the mutants, based on a given context-free grammar of the original artifact. The denotational semantics of $\mathscr{MuDeL}$ was used as a pseudo-oracle (in the sense discussed by Weyuker [28]) in the validation of the `mudelgen` [27]. And finally, by providing an abstract view of the mutations, $\mathscr{MuDeL}$ eases the reuse of mutant operators defined for syntactically similar 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, one can reuse the mutant operators that operate on the same construction, e.g., deleting statements, swapping expressions, and so on, on both languages. We have applied $\mathscr{MuDeL}$ and `mudelgen` with the languages C, C++ and Java and with the specification languages FSMs and CPNs. In particular, we used them in the context of Plavis project, which involves Brazilian National Space Agency. We observe that for languages with similar grammar, we could reuse not only the conceptual framework behind the mutation, but also the $\mathscr{MuDeL}$ mutant operators themselves.

Mutation testing demands several functionalities other than just generating mutants, e.g., test cases handling, mutant execution and output checking. Both $\mathscr{MuDeL}$ 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 having the most used languages as target languages. In fact, $\mathscr{MuDeL}$ and `mudelgen` are steps towards the implementation of such generic tools.

This paper is organized as follows. In Section 2 we discuss some related work and summarize the main features of mutant operators, highlighting the requirements for a description language for this specific domain. In Section 3 we present the $\mathscr{MuDeL}$ language and illustrate its main features. In Section 4 we show results from the application of the language we have made up to now, emphasizing the cases where we could effectively reuse mutant operator descriptions in different languages. In Section 5 we discuss relevant implementation aspects of the `mudelgen` system and depict its overall architecture. Finally, in Section 6 we make concluding remarks and point to further work.

2. Mutant operators

Mutation testing has been applied in several context, for several different languages. Therefore, mutant operators have been defined for those applications. The definitions are usually made in an *ad hoc* way, ranging from textual descriptions to formal

definitions. Notwithstanding, to the best of our knowledge, $\mathscr{MuDeL}$ is the first proposal to provide a precise language to describe mutant operators.

Mutation testing was first applied for the FORTRAN language [15]. DeMillo designed 22 mutant operators and developed the MOTHRA tool. The mutant operator descriptions were textual and heavily based on examples. Although the examples are very useful to illustrate the mutant operator, describing it by these means is ambiguous, and does not promote reuse.

Agrawal [14] proposed 77 mutant operators for the C language. The definition were based on the FORTRAN mutant operators. Most C mutant operators are basically a translation of the respective FORTRAN mutant operators. However, since the C language has a much richer set of arithmetic and logical operators, there are more mutant operators for the C language. These operators were implemented in the Proteum tool [29]. (Actually the mutant operators implemented by Delamaro et al. [29] are adapted versions of the Agrawal's ones.) Afterwards, Delamaro et al. [18] proposed mutant operators for testing the interfaces between modules in the C language, named interface mutation. The Proteum tool was extended with these operators, deriving the Proteum/IM [18].

Fabbri [30] investigated mutation testing concepts in the context of specification techniques for reactive systems. Mutant operators were designed for Petri nets, Statecharts and finite state machines (FSMs). Differently from the above approaches, those mutant operators were formally defined, using the same formalism of the corresponding technique.

Kim et al. [16] have proposed a technique named "hazard and operability studies" (HAZOP) [31] to systematically derive mutant operators. The technique is based on two main concepts. It first identifies in the grammar of the target language where mutation may occur and then defines the mutations guided by "Guide Words". They applied their technique to the Java language. Although the resulting operators do not significantly differ from previous works, the proposed methodology is an important step towards a more rigorous discipline in the definition of mutant operators.

From these examples, we can summarize the characteristics of mutant operators used in different context. Usually, from a single original artifact, a mutant operator will generate several mutants. For example, a mutant operator that exchanges a **while** statement into a **do–while** statement will generate as many mutants as the number of **while** statements in the artifact. In each mutant, a single **while** will be replaced by a **do–while**.

The number of mutants that can be generated from a particular artifact is very large. Considering an original artifact, any other artifact in the same language could be considered as a mutant, due to the informal and broad definition of "syntactical change" necessary to generate a mutant. To keep the number of mutants at a tractable level, only mutants with simple changes are considered. Roughly speaking, a change is considered simple when it cannot be decomposed into smaller, simpler changes. For that reason, to describe a mutant operator, usually only one change should be defined.

An important point that should be highlighted is that a change being simple does not mean it is straightforward. The syntax of the artifact should be taken into account, in order to generate syntactically valid mutants. Concerning this point, a mechanism based on simple text replacement is not enough. It is necessary to embody some mechanisms to guarantee that the mutants are also valid artifacts in the original language. The pattern replacement, which is typical in transformational languages, is more suitable in this context.

Sometimes the single logical change implies in changing more than one place in the artifact. For example, a mutant operator for exchanging two constants must indicate that two different but related changes, one in each place where the exchanging constants appear. Moreover, in some cases, although being treated as a single entity, the mutant operator involves different changes in different places. Therefore, it is necessary to be able to relate these different changes into the mutant operator.

Mutations can be classified into two major groups: context-free mutations and context-sensitive ones. Context-free mutations are those that can be carried out regardless the syntactical context in which the mutated part occurs. Conversely, context-sensitive mutations depend upon the context, e.g., the variables visible in a specific scope. Most mutant operators in literature [3,6,14,17] involve context-free mutations. Even for a context-sensitive language, there can be context-free mutations. An example of context-free mutations is the change of "`x = 1`" by "`x += 1`", since wherever the first expression is valid, so is the second. However, the change of "`x = 1`" by "`y = 1`" is context-sensitive, since the second expression will not be valid unless `y` has the same declaration status as does `x`. To tackle this 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.

3. $\mathscr{MuDeL}$ language

Based on the characteristics of mutants, we designed a language to allow the definition of mutants in a way as easy, language-independent and natural as possible. However, due to pragmatical issues, we have taken some design decisions that trades off between the goals listed above and the possibility of implementing of an efficient supporting tool. Therefore, $\mathscr{MuDeL}$ does not provide a completely language-neutral mechanism for describing and implementing mutant operator. Indeed, the syntax of the target language should be somewhat embodied in the mutant definition. $\mathscr{MuDeL}$ language is, thus, a mixed language that brings together concepts of both the transformational and the logical paradigms. From the transformational paradigm it employs the concept of pattern matching and replacement. The transformational language that is closest to $\mathscr{MuDeL}$ is TXL [32,33]. TXL has both matching and replacement operations. However, TXL works in a one-to-one basis and has a imperative-like control flow, making unnatural to describe mutant operators. Instead, the control flow of $\mathscr{MuDeL}$ is inspired in Prolog's. The most important similarity of $\mathscr{MuDeL}$ and Prolog's is, however, the way a mutant operator definition is interpreted. Like a Prolog clause,

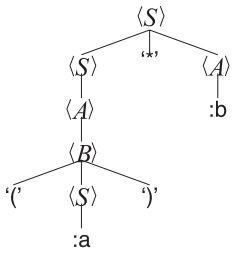

Fig. 1. The pattern tree for '`( :a ) * :b`'. The types of '`:a`' and '`:b`' have been declared as ⟨*A*⟩ and ⟨*B*⟩, respectively.

a $\mathcal{MuDeL}$ definition can be thought of as a predicate. A mutant should satisfy the "predicate" of a mutant operator definition in order to be a mutant of this operator w.r.t. the respective original artifact. However, like a `findall` predicate in Prolog, the $\mathcal{MuDeL}$ definition can be used to enumerate all mutants that satisfy it. This is made by the `mudelgen` system, described in Section 5.

3.1. Basic notations

In order to be able to handle different kinds of artifacts, we should choose an intermediate format to which every artifact can be mapped. Assuming that most artifacts can be thought of as elements of a language defined by a context-free grammar, the use of syntax tree has an immediate appeal [34]. Therefore, in the $\mathcal{MuDeL}$ language, every artifact, either a program or a specification, is mapped into a syntax tree. The mapping is carried out by parsing the artifact based on a context-free grammar of the language. The syntax tree can be handled and modified in order to represent the mutations. It is thus necessary a way to describe how the syntax tree must be handled.

We define a set $\mathcal{M}$ of *meta-variables*[1] and extend the syntax tree to allow 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 the 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. Fig. 1 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 artifact, i.e., a meta-variable can only occur where a non-terminal of its type also could.

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 ⟨*A*⟩. In most cases, such a simple notation will not be enough to specify pattern trees. One can use a more elaborated pattern notation, instead. The non-terminal root symbol is placed in squared brackets, as before, but following it, in angle brackets, a sequence of terminal symbols and meta-variables is included. For example, the pattern tree in Fig. 1 is denoted by `[S<( :a ) * :b>]`. Note that inside the angle brackets the grammar of the artifact, 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. Therefore, instead of being just a language, $\mathcal{MuDeL}$ is indeed a *meta-language*, in that a $\mathcal{MuDeL}$'s definition is valid or not w.r.t. a given source grammar. In other words, given a source grammar of an artifact language, we can instantiate $\mathcal{MuDeL}$ language for that grammar. The source grammar determines the form and the syntax of the pattern trees.

The unification of a tree and a pattern is in the kernel of any transformational system. In the unification, two pattern trees *c* and *m* are taken and an attempt to unify them is done. The unification can either fail or succeed. In case of success, the meta-variables in the pattern trees are accordingly bound to respective tree nodes, in a way that makes them unrestrictly interchangeable. In case of failure, no meta-variable binding occurs. The unification algorithm is similar to Prolog's one [24]. Fig. 2 shows an example of a successful unification. The dashed line indicates the meta-variable bindings.

3.2. Operator structure

A mutant operator definition has three main parts: operator name, meta-variable declarations and body. The operator name declaration comes first. This name is just for documentation purposes and has no impact in the remaining parts of the definition. 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, which is its type. The last section, enclosed by the keywords `begin` and `end operator`, is the body of the operator, which

[1] We chose the term *meta-variable* instead of the term *variable*, which has a particular meaning in most language to which $\mathcal{MuDeL}$ can be applied.

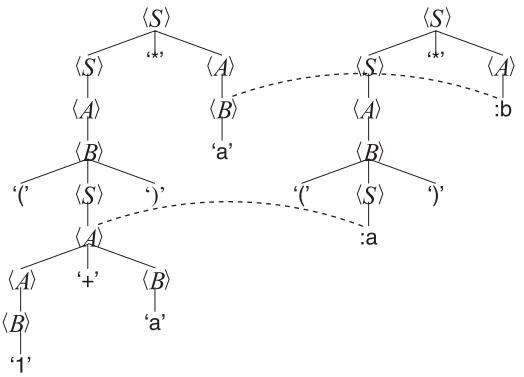

Fig. 2. An example of unification.

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

Fig. 3. A simple mutant operator. For every statement in the program, a mutant is generated by "deleting" the statement.

is a compound mutation operation (explained later). This operation will be executed w.r.t. the syntax tree of the original artifact. Fig. 3 presents a mutant operator definition, 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 as its body, 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. Observe that there is no explicit indication of which node should be considered by the replace operation, which, in this case, implies that the whole tree should be used.

The body of a mutant definition written in $\mathscr{MuDeL}$ is composed by a combination of *operations*. The syntax of $\mathscr{MuDeL}$'s body part can be divided into operations, combiners and modifiers. An operation can be thought of as a predicate that either modify the syntax tree or control the way the remain operations act. The operations can be joined by combiners. The behavior of an operation can be altered by modifiers.

If a set of operations must be used in several different points in a mutant operator, it is possible to declare a *rule* with these operations and invoke the rule wherever necessary. Rules can be thought of as procedures of conventional programming languages. In this way, mutant operators can be defined in a modular way. Rules can be defined in a separated file and imported in the mutant operator, allowing to reuse similar operations among a set of related mutant operators.

3.3. Operations

An operation is a particular statement about how to proceed in the generation of a particular mutant. An operation takes place in a particular state, which is formed by the current syntax tree. Every operation, being it simple or compound, can result in zero or more alternative syntax trees. If it results in zero alternative syntax tree, we say that the operation result in failure, i.e., that it fails.

3.3.1. Replace operation

The replace operation is the most important one in $\mathscr{MuDeL}$ language, since it is responsible for altering the original syntax tree into the mutated one. It requires three arguments: the tree *c* to be altered, the pattern tree *r*, that is to be unified with *c*, and the pattern tree *b*, that will replace *c* in the case of a successful unification of *c* and *r*. Both *r* and *b* can contain meta-variables that allow to use parts of *c* in the replacement.

The syntax of a replace operation is

```
c @@ replace r by b
```

where *c* must be a meta-variable.

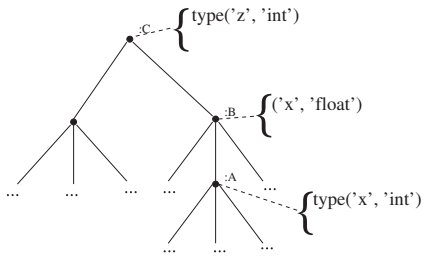

Fig. 4. Facts in the syntax tree.

3.3.2. Match operation

The matching operation can be used to select where the replacement is applied. The matching operation takes two arguments: a tree *c* and a pattern tree *m*. It tries to unify *c*, which must be a meta-variable, and *m*, binding meta-variables in *m* if necessary. The bindings are still active after the unification, allowing to select parts of *c* to be further handled.

3.4. Assert and consult operations

Most of the mutations can be made in a context-free basis. This is true due to fact that if the original artifact is syntactically valid, the mutant operator can safely rearrange some part of it and ensure the mutant is also syntactically correct. However, there are some mutant operators that require some information that comes from the context in which the mutated parts are located. For instance, when exchanging a variable by another one, it is necessary to check whether their declared types are compatible. Strictly speaking, this could be made with the operations, modifiers and combiner described. Nonetheless, such a way of definition will be quite awkward. We tackle this problem by enriching the syntax tree with *attributes* [35]. The attributes are a set of tuples that has a name and a set of values and is associated to a node in the syntax tree. The attributes are calculated and stored when the syntax tree is built. (See Section 5 for a discussion on how the attributes are calculated.)

To access the values of the attributes, $\mathscr{MuDeL}$ has the `consult` operation. The consult operation takes an starting tree node *c*, an attribute name *n*, and a list of meta-variables or pattern trees, which represent the arguments of the attribute. The operation will look for any tuple with name *n* in the tree node *c*. If it finds any, it will try to unify the list of arguments with the list of arguments in the tuple. Each tuple that successfully matches the list of arguments will produce an alternative state. However, when there is no tuple in *c* with name *n*, the consult operation will recursively search in the parent node of *c*, until a node with such a tuple is found or it has already searched in the root of the tree (that has no parent). Observe that this upward search embodies the way context information is usually dealt with. It also allows the correct dealing with a scope of most typed languages, in which the attributes of an entity can be overridden in an inner scope. The `consult` operation can be compared to the Prolog `consult` predicate. However, the Prolog's `consult` predicate uses a single global base of facts, while in $\mathscr{MuDeL}$ the facts are scattered over the tree and are searched in a hierarchical way. The operation can be negated, analogously to the matching operation. Fig. 4 illustrates how attributes are stored and retrieved in syntax trees. We annotate in each which tuples are defined. If the operation

```
:C @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will fail, since there is no fact about 'x' in `:C`. However, if the same operation is executed with `:v` unified to 'z', it will succeed with the unification of `:t` to 'int'. If the operation

```
:B @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will succeed with the unification of `:t` to 'float'. If the same operation is executed with `:v` unified to 'z', it will succeed with the unification of `:t` to 'int', since this fact is stored in the parent of `:B`. If the operation

```
:A @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will succeed with the unification of `:t` to 'int'. Observer that the facts in `:B` were overridden.

In most of the cases, the attributes that are consulted are stored in the tree when it is built (see Section 5). However, sometimes, it may be useful to also be able to store tuples in the tree. This is made with the `assert` operation. It takes a context tree *c*, an attribute name *n*, a list of meta-variables or patterns, which represents the arguments of the tuple, and a list of patterns that represent where the tuple should be stored. The operation will search upwards from *c* for a tree node that matches any of the patterns. If it finds such a node, the tuple is stored in it. Otherwise, the tuple will be stored in the root node.

3.4.1. Donothing, abort and cut operations

We include some atomic operations that enhance the control of generation of the mutants. They control the set of alternative states the next operations will deal with. The `donothing` operation, as its name suggests, does nothing at all. It succeed exactly once. It can be thought of as a placeholder for situations where an operation is necessary but no effect is indeed required. It is similar to the `true` predicate of Prolog.

```
:p @@ replace [S< while( :e ) :s >]
      by      [S< do :s while ( :e ); >]
```

Fig. 5. Replace operator.

a

```
while (a < b) {
  if (a % 2 == 0) {
    a ++;
  } else {
    a += 3;
  }
}
```

b

```
do {
  if (a % 2 == 0) {
    a ++;
  } else {
    a += 3;
  }
} while (a < b);
```

Fig. 6. An example of the application of operation in Fig. 5.

The `abort` operation will ignore any alternative state. It always results in failure and, therefore, can be used (in conjunction with the combiners) to avoid generated some mutants. It is similar to the `false` predicate of Prolog.

The `cut` operation will prune the set of alternative states, in such a way that only the first alternative state will be considered. It is similar to, and was inspired by, the `!` of Prolog.

3.5. $\mathcal{MuDeL}$ combiners

Two or more operations can be combined into a compound operation using the combiners `;;` and `||`. They were inspired in the Prolog operators comma (,) and semi-colon (;).

The first combiner is the *sequence* one, which is represented by `;;` in the $\mathcal{MuDeL}$ syntax. The compound operation `a;;b` incorporates the effects of both `a` and `b`. Every time the operation `a` results in success, the operation `b` is applied. As a side-effects, if the operation `a` does not succeed, the operation `b` will be ignored.

The second combiner is the *alternative* one, which is represented by `||` in the $\mathcal{MuDeL}$ syntax. The compound operation `a||b` indicates that both `a` and `b` are alternative operations for the same purpose. Therefore, the results of either one can appear in a mutant. Actually, the compound operation succeeds every time the operation `a` does and every time the operation `b` does.

The combiners can be used with more than two operations. For instance, we can join three operations as in `a;;b;;c`. Moreover, both combiners can be used together. In this case, the combiner `;;` has a higher precedence than the combiner `||`. The operations can be grouped with double parenthesis to overpass the precedence. For instance, in the compound operation `((a||b));;c`, the operation `c` will be applied to the alternative syntax trees resulting from the operations `a` and `b`.

3.6. $\mathcal{MuDeL}$ modifiers

There are two modifiers that can be applied to an operation. The negation modifier is used to "invert" the result of an operation. It is syntactically represented by a `~` placed in front of the modified operation. Every operation in $\mathcal{MuDeL}$ can result in either a failure or a success. The precise meaning depends on the specific operation on which it is applied. For instance, the `match` operation results in a success if it can unify its operands, and results in failure otherwise. When modified with `~`, a unification will be considered a failure, while the inability to unify will be considered successful.

The *in depth* modifier is used to indicate that the modifier operation should be applied not only to the context tree, but also to every of its subtrees. For instance, when applied to a `match` operation, the unification will be tried with the context tree and with each of its subtrees. Whenever a unification is successful, the `match` operation will result in success and a mutant will be generated. For the `replace` operation, the effects of the modifier is similar. The replacement will be made not only in the context tree, but also in every of its subtrees, in turn. It is important to note that each replacement will take place in the original tree, i.e., after each replacement, the tree is restored to the original one before the next replacement be searched.

3.7. Usage examples

In this section, we illustrate the usage of elements of $\mathcal{MuDeL}$ syntax. We present only the operator body, not including the operators name and the meta-variable declaration sections, once they can be inferred from the operations. Moreover, we give only an informal definition of the semantics of each operation. A formal definition can be found elsewhere [27]. All the examples describe mutant operators for the C language.

An example of a replace operation is presented in Fig. 5, that replaces a **while** statement (matched in Line 1) by a **do–while** one (Line 2). This is the objective of the SWDW mutant operator defined by Agrawal [14].

Assuming that `:p` is bound to the syntax tree of the fragment of C code in Fig. 6(a), after the application the operation in Fig. 5, the code will be replaced by that in Fig. 6(b).

a

```
if (a > 0) {
  while (a < b) {
    a *= 3;
  }
}
```

b

```
while(...) {
  ...
}
while(...) {
  while(...) {
  }
}
```

Fig. 7. An example of **while** statements that will not be replaced by the operation in Fig. 5.

```
:p @@ * replace [S< while( :e ) :s >]
        by      [S< do :s while ( :e ); >]
```

Fig. 8. The replace operation modified by *.

```
int doOpen() {
  if (isClose()) {
    return open();
  } else {
    return 1;
  }
}
```

Fig. 9. A C function example.

```
1 :f @@ match [FD< int :id () :s >]
```

Fig. 10. An example of the matching operation.

```
1   :f @@ * match [FD< while( :e ) :s >]
2 ;;
3   :e @@ * replace [S< :id >]
4            by      [S< 0 >]
```

Fig. 11. A usage example of the combiner ; ;.

Suppose now that :p is bound to the C code in Fig. 7(a). In this case the operation in Fig. 5 will not be applicable, since the **while** statement will form a sub-tree of the whole statement, and, thus, the pattern of the replace operation will not unify to it. Another situation that should be dealt with is illustrated in the C code in Fig. 7. In this case, we have three **while** statements.

To properly deal with this situation, we can modify the replace operation with the modifier *. The meaning of a replace operation with the modifier * is that every successful matching of *b* with *c* itself or any of its subtrees will produce a mutated tree. Indeed, we can think of this modified operation as producing alternative states, and each of such states will have its own execution flow and eventually producing a mutant. Therefore, a more adequate mutant for the SWDW is the one presented in Fig. 8.

Suppose that the meta-variable :f is bound to the code in Fig. 9. Then, after the application of the matching operation in Fig. 10, the meta-variable :s will be bound to the body of the function. Observe that the pattern will match a function with no arguments that returns an int value.

The compound operation in Fig. 11 will replace every variable in the control expression of a **while** statement to 0. It is important to note that, for every such a control expression, the matching will produce an alternative state and the replace operation in Line 3 will be applied to each one, possibly generating more alternative states by itself. This example illustrates the usage of the match operation to constrain the context in which the replacement should be applied. In this case, the operator was designed to be applicable only to **while** statement control expressions. Suppose now that one wants the replacement to be applied to control expression of every iterate statement, i.e., every **while**, **do–while** or **for** statement. In other words, we want to join the set of

```
1  ((
2    :f @@ * match [FD< while( :e ) :s >]
3  ||
4    :f @@ * match [FD< do :s while( :e ) ; >]
5  ||
6    :f @@ * match [FD< for ( :init: ; :e :inc ) :s >]
7  ))
8  ;;
9    :e @@ * replace [S< :id >]
10            by      [S< 0 >]
```

Fig. 12. A usage example of the combiner ||.

```
:f @@ * ((
  match [FD< while( :e ) :s >]
||
  match [FD< do :s while( :e ) ; >]
||
  match [FD< for ( :init: ; :e :inc ) :s >]
))
;;
  :e @@ * replace [S< :id >]
          by      [S< 0 >]
```

Fig. 13. An example of the usage of parenthesis to factor out common modifiers.

```
  :s @@ * replace [E< :id >]
          by      [E< 0 >]
;;
  :id @@ consult type with :id [T< int >]
```

Fig. 14. Example of `consult` operation.

```
  * match [ID< :id1 >]
;;
  * replace [ID< :id2 >]
    by      [ID< :id1 >]
;;
  :id1 @@ ~ match [ID< :id2 >]
```

Fig. 15. Example of operations that exchanges an identifier by every other identifier.

alternative states of three matching operations. This can be done with the alternative combiner, which is represented by || in the $\mathcal{MuDeL}$ syntax. For instance, the compound operation in Fig. 12 will achieve the objective. (The parenthesis are necessary because ;; has a higher precedence than ||.)

The parenthesis can also be used to avoid explicitly declaring the context tree in every operation, as well as the * modifier. Therefore, the compound operation in Fig. 12 is equivalent to the compound operation in Fig. 13. There is, however, a small difference between both w.r.t. the efficiency and the order in which the alternative states (and, hence, the eventual mutants) are produced. While in Fig. 12 :f is traversed three times, since for each match operation starts from the root node of the :f syntax, in Fig. 13 :f is traversed only once.

The other two basic operations (namely, **assert** and **consult**) are related to context-sensitive mutants. The `consult` operation in Fig. 14 is used to ensure that only identifiers with the `int` attribute is mutated to `0`.

To illustrate the use of the assert operation, consider a mutant operator that exchanges each identifier by each distinct identifier in the artifact. (For sake of simplicity, we assume that this mutation can be carried out without taking context into consideration.) Firstly, consider the operator in Fig. 15, which will exchange each identifier matched in Line 3 by the identifier

```
  * match [ID< :id1 >]
;;
  ~ consult used with :id1
;;
  assert used with :id1
;;
  * replace [ID< :id2 >]
    by      [ID< :id1 >]
;;
  :id1 @@ ~ match [ID< :id2 >]
```

Fig. 16. Example of usage of `consult` and `assert` operations to avoid employing the same identifier more than once.

```
:f @@ * ((
  match [FD< while( :e ) :s >]
||
  match [FD< do :s while( :e ) ; >]
||
  match [FD< for ( :init: ; :e :inc ) :s >]
))
;;
  :e @@ * replace [S< :id >]
           by      [S< 0 >]
;;
  cut
```

Fig. 17. Usage of `cut` operation.

matched in Line 1 that are not equal to each other (ensured by the negated matching in Line 6). If the same identifier occurs in more than one location, the match in Line 1 will produce an alternative state for each one, and the replacement will generate several identical mutants. We canavoid this situation with the usage of `consult` and `assert` operations, as illustrated in Fig. 16. In this way, the `consult` operation in Line 3 ensures that there is no tuple for the `used` attribute with the `:id1` value. Then, if so, the assert operation stores such a tuple in the root node (since no context pattern was furnished).

The `cut` operation can be used to prune the set of alternative states that the previous operations might have generated. It was introduced in $\mathscr{MuDeL}$ language for sake of completeness, since the other operations are inspired in Prolog, and this language has the cut operator (!), whose purpose is similar. An example of the usage of cut is presented in Fig. 17, which equals the example in Fig. 13, expect for the `cut` operation added in the end. The effect is that after the `cut` operation is executed, any pending alternative states are forgotten, i.e., at most one mutant will be generated.

In Fig. 18 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. This kind of mutant is usually necessary when branch coverage is required. Observe that, in a C program, changing the control expression into 0 has the effect of iterating the body of the **while** statement exactly once, whereas changing the control expression into 1 converts the **do–while** into an infinite loop. Fig. 19(a) presents a simple C program and Figs. 19(b)–(e) present the mutants that will be generated for this program with the SDWE operator.

The replacing operation in Lines 5 and 6 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 8–20 makes changes in this control expression. Observe that the context pattern declaration in Line 8 affects the whole group, and, consequently, every operation therein.

The (negated) matching in Line 9 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 replacement in Lines 11 and 12, and a mutant is generated. Note that these two operations compose a sequence, which is part of an alternative list. Then, the next alternative is tried, in this turn w.r.t. the expression 1. Finally, the operation in Line 19 is tried and a mutant is generated only with the replacement of Line 5.

Analyzing how the mutants are generated in this example illustrates the way $\mathscr{MuDeL}$ processes a mutant operator definition. The replacing operation (Lines 5 and 6) is marked with the *in depth* modifier and, therefore, the whole program syntax tree will be scanned, looking for nodes that match the respectivepattern and changing them accordingly. The replacing operation and the group of operations in Lines 8–20 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 alternatives: the first alternative is in Lines 9–12; the second one is in Lines 14–17; and the last one is in Line 19. Only the effects (if any) of one of these alternatives will be included

```
operator SDWE
  var :e [expression]
      :s [statement]
begin
  * replace [statement< while ( :e ) :s >]
    by      [statement< do :s while ( :e ) ; >]
;;
  :e @@ ((
      ~ match [expression< 0 >]
    ;;
      replace [expression]
      by      [expression< 0 >]
  ||
      ~ match [expression< 1 >]
    ;;
      replace [expression]
      by      [expression< 1 >]
  ||
    donothing
  ))
end operator
```

Fig. 18. A multi-purpose **while** mutant operator.

in a particular mutant. For instance, Mutants #1 and #2 in Figs. 19(b)–(c) are generated by replacing the outermost **while** of the program in Fig. 19(a) and applying the first and the third alternatives, respectively. (Observe that the second alternative does not generate a mutant, since the operation in line 14 does not succeed.) On the other hand, Mutants #3–#5 in Figs. 19(d)–(f) are generated by replacing the innermost while and applying each of the alternatives, respectively.

4. Applying $\mathscr{M}u\mathscr{D}e\mathscr{L}$

The usefulness of $\mathscr{M}u\mathscr{D}e\mathscr{L}$ can be measured by its suitability in defining mutants, which is its primary goal; in allowing the reuse of mutant operators in different languages; and in generating a mutant generator prototype module that can be evolved and incorporated into a mutation-testing environment. Currently, we have already described mutant operators for (i) specifications written in colored Petri nets (CPNs) and FSMs, (ii) for the functional language standard meta-language (SML), and (iii) for traditional languages such as C, C++ and Java.

In this section, we present our experience using $\mathscr{M}u\mathscr{D}e\mathscr{L}$, and bring some evidence of its usefulness. However, more studies are necessary in order to soundly validate the language. Moreover, there is a lack of feedback from other research groups, and a thorough validation would involve the use of the language by others.

FSM, CPN and SML mutant operators: We used $\mathscr{M}u\mathscr{D}e\mathscr{L}$ to define mutants for specifications written in FSMs [36] and CPNs [37]. In the case of FSMs, we use $\mathscr{M}u\mathscr{D}e\mathscr{L}$ to describe the nine mutant operators defined by Fabbri et al. [38]. Using the `mudelgen` system (described in the next section), we were able to use the mutant operators within the Plavis/FSM environment, which are being used in experiments in Brazilian National Space Agency, in the scope of a project supported by CNPq and CAPES-Coffecub.[2] In the case of CPNs, we observed that the language was useful to allow a rapid prototyping and experimentation with different kinds of mutants. In the whole, 29 mutant operators were defined and used in Proteum/CPN tool [37]. It is important to mention that CPNs annotation language is based on SML and we could reuse some common parts of the mutant operator descriptions in both languages [39].

C and C++ mutant operators: For C language, we described the 77 mutant operators proposed by Agrawal [14], which are implemented in Proteum/C tool. Next, we adapted these operators and described similar mutant operators for C++. We realized that, by carefully designed the grammar and the $\mathscr{M}u\mathscr{D}e\mathscr{L}$ definition, we could reuse 65 operators, nearly 85% of them. To illustrate how this was possible, consider again the operator in Fig. 5. Examining the definition, we can observe that the only relation between the operator and the language of the artifact is in the patterns. Even in the patterns, only the types of the patterns and of the meta-variables and the sequence of terminal symbols are relevant. Therefore, the same definition can be applied both for C and C++, provided that the respective grammars agree in these points: (i) the name of the relevant non-terminal symbols

[2] http://www.labes.icmc.usp.br/plavis/.

a

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

b

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

c

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

d

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

e

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

f

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

Fig. 19. (a) Original program. (b)–(e) Mutants generated by operator in Fig. 18. The mutated parts of the code are highlighted.

(that define the types available) and (ii) the sequence of terminal symbols that appear in the relevant non-terminal productions. Observe that not the whole sequence of terminal symbols should be the same. Rather, only the terminal symbols that are relevant to the $\mathscr{MuDeL}$ definition. In the SDWD example, for the mutant definition to be applicable, it is necessary that both grammars have a non-terminal symbol S and that there is a derivation from S to '`while`' '`(`' ⟨*E*⟩ '`)`' ⟨*S*⟩ and to '`do`' ⟨*S*⟩ '`while`' '`(`' ⟨*E*⟩ '`)`' '`;`'.

Java mutants: We have also carried out a study, in which we try to apply the mutant descriptions for C and C++ to the Java language. Since the grammar of these languages are similar, we could observe that 31 mutant operators could be reused for Java. The results of the application to C, C++ and Java are summarized in Fig. 20.

We have investigated how $\mathscr{MuDeL}$ can be used with mutant operators that are more semantic-driven. We have described the class mutant proposed by Kim et al. [40,41]. Some of those operators are related to the semantics of inheritance, overloading and overriding concepts, which varies from one OO language to another. Those complex operators can only be described with complex $\mathscr{MuDeL}$ code. Indeed, the complexity is inherent to the operators and, to our knowledge, their definition could only made simpler if we hide the complexity in a more complex operation of the language. The same situation occurs with any language. There is a trade-off between the simplicity of the operations and its ability to handle complex mutants. The complexity of these kinds of mutants comes from the underlying semantics of those languages. For instance, let us consider CM operators defined by Kim et al. [41]. From the 20 mutant operators, we could describe easily 10 of them, since they require only syntax driven changes. Other five could be described, provided that some semantic information is collected and is available to consult operations. However, this semantic should be coded, anyway, and to adequately capture the semantics of OO languages is not an easy task and is still open issue in the programming language semantics research. Unfortunately, we could not find an easy way to tackle this problem, either. The remaining five ones might be described, depending on what exactly the authors mean. For instance, FAR operator is defined as "Replace a field access expression with other field expressions of the same field name". Unfortunately, it is not clear what "a field access expression" exactly means. Other example of ambiguity is the definition of AOC operator. In Kim et al. [41], the authors defined it as "Change a method argument order in method invocation expressions".

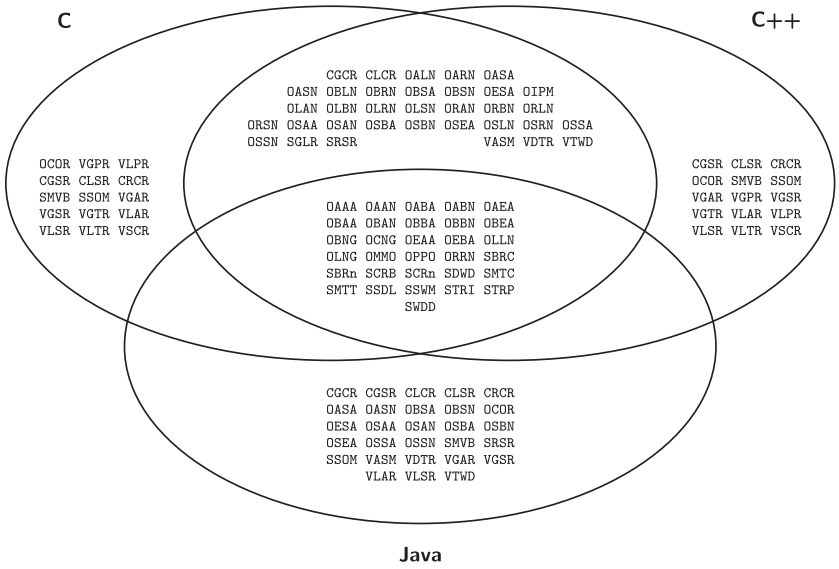

Fig. 20. Mutant definition reuse for C, C + + and Java.

It is not clear how many mutants can be generated from a method with more than two arguments. The possibilities are: (i) one mutant for each permutation of the arguments (i.e., exponentially many mutants); (ii) one mutant for every shift, in the same vein as Agrawal's SSOM mutation operator. In Kim et al. [40], the authors give a little bit more explanation about the operator and provide an example, neither of which clarify the point. These ambiguities evidence the necessity of a formal definition of mutant operator.

5. mudelgen

In order to be able to process the $\mathscr{MuDeL}$ descriptions, we implemented the `mudelgen` system. In this section we discuss its main implementation aspects. Suppose that we are interested in describing mutant operators for a language *L*. The first step is to obtain a grammar *G* for *L*. When `mudelgen` is input with *G*, it produces a program `mudel.G`. This program can then be run with a mutant operator definition *OD* and an artifact *P*. After checking whether both *OD* and *P* are syntactically correct w.r.t. the input grammar *G*, a mutant set *M* is generated.

To manipulate the `mudelgen` input grammar, we use `bison` and `flex`, which are open source programs similar to, respectively, `yacc` and `lex` [42]. Although these tools ease the task of manipulating grammars, they, on the other hand, restrict the set of grammars that `mudelgen` can currently deal with to LALR(1) grammars [34,35,42]. The grammar input to `mudelgen` is provided in two files: the `.y` and the `.l`. The `.y` file is the context-free grammar, written in a subset of `yacc` syntax [42]. The `.l` file is a lexical analyzer and gives the actual form of the terminal symbols of the grammar and it is encoded in a subset of the `lex` syntax [42]. The attribute values are attached to the tree nodes with special C functions put in the semantic action of the productions of `.y`. For instance, the function `assertFact` can be used to store an attribute value in a way similar to the `assert` operation.

The `mudelgen` is divided into two parts: one part with the elements that depend on the input grammar and the other one with elements that do not. Fig. 21 depicts how these parts interact and illustrates the overall execution schema of `mudelgen`. The grammar-dependent part is actually composed by three modules, which are executable programs: `treegen`, `opdescgen` and `linker`. The grammar-independent part is embodied in the `Object Library`. The major portion of the `Object Library` is devoted to the so-called MuDeL Kernel, which is responsible for interpreting the mutant operator definition and manipulating the syntax tree accordingly. The remaining units in the `Object Library` allow the communication between the MuDeL Kernel and the external modules $\mathscr{MuDeL}$ Animator and DS Oracle, described later.

The units that depend on the grammar are built either by `treegen` or by `opdescgen`. Module `treegen` analyzes *G* and generates the units: (i) `STP` (syntax tree processor), which is responsible to syntactically analyze a source product *P* into a syntax tree and (ii) `Unparse`, which is responsible to convert the mutated syntax trees into the actual mutants. Module `opdescgen` analyzes *G* and generates the unit `ODP` (operator description processor), which analyzes a mutant operator description *OD* w.r.t. *G* and generates an abstract representation of how to manipulate the syntax tree in order to produce the mutants. Finally, the `linker` module will link all these grammar-depending units and the appropriate portion of the `Object Library` and generate the program `mudel.G`.

The program `mudel.G` is input with a source product *P* and a mutant operator definition *OD*. These input data are processed by `STP` and `ODP`, respectively, and handled by MuDeL Kernel. During its execution, MuDeL Kernel will generate one or more

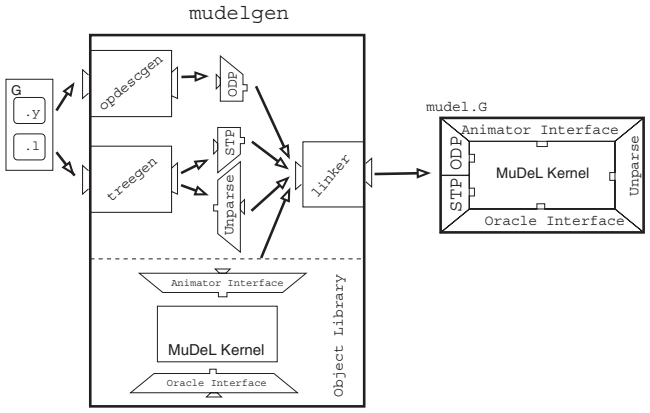

Fig. 21. mudelgen execution schema.

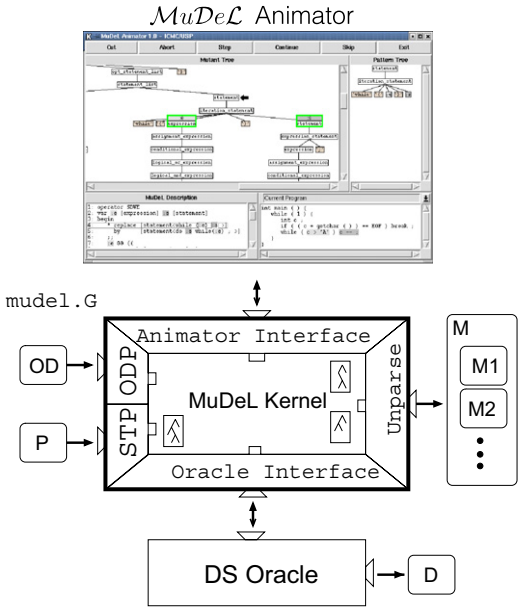

Fig. 22. Execution of `mudel.G`.

mutated syntax trees, which are processed by `Unparse` in order to generate the actual mutants. `Unparse` can output the generated mutants in several formats. Currently, the mutants can be (i) sent to standard output; (ii) restored in SQL databases (e.g., MySQL); or (iii) written to ordinary files (each mutant in a separate file). Optionally, the DS Oracle can be used to check whether the mutants were correctly generated (see Section 5.1). The execution of the program `mudel.G` can be visually inspected with the $\mathcal{MuDeL}$ Animator (see Section 5.2). The overall execution schema of `mudel.G` is depicted in Fig. 22.

5.1. Denotational semantics-based oracle

The number of mutants generated is often very large and manually checking them is very costly and error-prone. Therefore, the validation of `mudelgen` is a hard task, mainly due to the amount of output which is produced. To cope with this problem, we adopted an approach that can be summarized in two steps. Firstly, we employed *denotational semantics* [25] to formally define the semantics of $\mathcal{MuDeL}$ language [27]. Secondly, supported by the fact that denotational semantics is primarily based on lambda calculus, we used the language SML [43], which is also based on lambda calculus, to code and run the denotational semantics of $\mathcal{MuDeL}$. We implemented an external module DS Oracle that can be run in parallel with `mudel.G` through the `Oracle Interface` in a *validation* mode. When invoked, the DS Oracle receives the information about a mutant operator *OD*

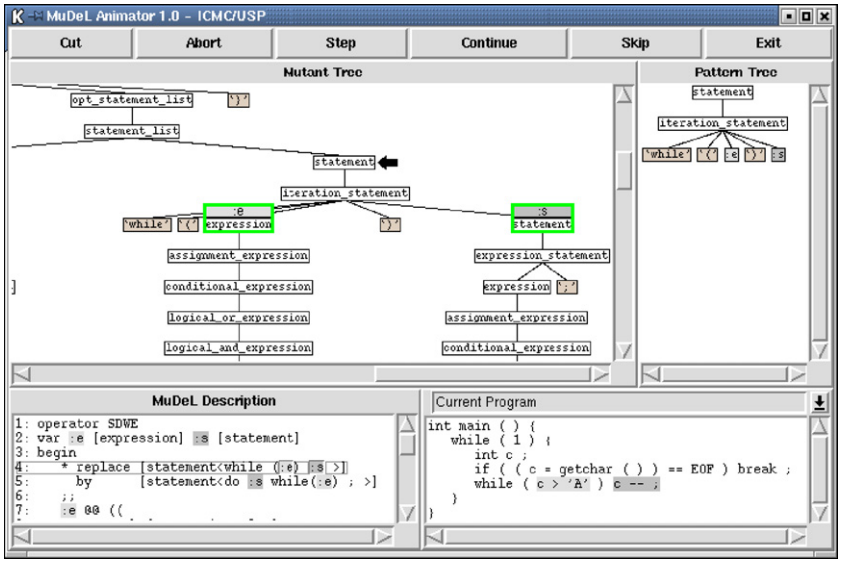

Fig. 23. $\mathcal{MuDeL}$ Animator main window.

and derives a denotational function μ (in the mathematical sense) that formally defines the semantics of *OD*. Then, the DS Oracle reads the information about the source product *P* and the set of generated mutants *M*. The mutants in *M* are compared with the mutants defined by μ. Any identified difference is reported in the discrepancy report *D*.

It is important to remark that the *validation* mode has no usefulness for users interested in `mudelgen`'s functionalities, since it brings no apparent benefit. However, it is very useful for validation purpose, since it improves the confidence that the mutants are generated in the right way. Nonetheless, from a theoretical viewpoint, there is a possibility that a fault in the implementation be not discovered, due to the fact that the SML implementation may also possess a fault that makes it produces the same incorrect outputs. However, the probability that this occurs in practice is very small. Both languages (i.e., C + + and SML) are very different from one another. Moreover, the algorithms and overall architectures of both implementations are very distinct. While we employed an imperative stack-based approach in C + +, we extensively used continuation and mappings [25] in SML. Consequently, it is not trivial to induce the same kind of misbehavior in both implementations. In other words, although none of them is fault free, the kind of faults they are likely to include is very distinct. With this consideration, we conclude that the use of denotational semantics and SML was a powerful validation mechanism for `mudelgen`.

5.2. $\mathcal{MuDeL}$ animator

We have also implemented a prototyping graphical interface—called $\mathcal{MuDeL}$ Animator—for easing the visualization of the execution of a mutant operator. $\mathcal{MuDeL}$ Animator was implemented in Perl/Tk and currently has some limited features that allows inspecting the log of execution, without, however, being able to interfere in the process. Since $\mathcal{MuDeL}$ Animator enables us to observe the execution of a mutant operator definition, it is very useful not only for obtaining a better understanding of the $\mathcal{MuDeL}$'s mechanisms, but also for (passively) debugging a mutant operator.

Fig. 23 presents the main window of $\mathcal{MuDeL}$ Animator, where the example of Fig. 18 was loaded and is being executed. 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:

$\mathcal{MuDeL}$ description: In the left bottom area, $\mathcal{MuDeL}$ Animator presents the mutant operator definition. 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 left top 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. Meta-variable bindings are presented by including the names of the meta-variable above the respective tree nodes.

Current product: In the right bottom 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.

Pattern tree: In the right top area, $\mathcal{MuDeL}$ Animator shows the tree of the pattern currently active (i.e., in the current line) in the $\mathcal{MuDeL}$ description area.

Since $\mathcal{MuDeL}$ 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 $\mathcal{MuDeL}$'s mechanisms, but also for (passively) debugging a mutant operator.

5.3. Operational aspects

We have designed `mudelgen` to generate a module for each particular language. These modules might be used as a standalone tool, or as an component within a larger, more complete upper-level environment. Therefore, we delegate some common tasks of managing mutants to this upper-level environment, such as (i) preparing the source code; (ii) selecting parts of the source code to which the operators should be applied; (iii) selecting which of the generated mutants should be used or discarded, and so on. We decided to keep these tasks to the upper-level as a way of increasing flexibility. In this way, the module could be used in different context and with different purpose, such as constraint mutation [44], essential mutation [11] and so on. However, this upper-level environment is very important to make the application of mutant testing feasible. For instance, the module `mudelgen` will apply the mutant operator to the whole source code that is provided as input. For large source codes, this could be impractical or even impossible.

Besides being generated, the mutants must be executed in some way, in order to collect the results and decide whether a mutant was killed by a test set or would stay alive. In general, the execution of mutants can be very costly. In the particular case of mutants of programs, it may be necessary to compile and execute every mutant. To tackle this problem, some researchers have proposed alternative schemas of execution. For instance, Untch et al. [45] use mutant schemata. Several mutants are put in a single generic source, which is parameterized to behave like any of the individual mutants. In this way, only one compilation for each schemata is necessary. A similar approach is employed by Delamaro et al. [18] in Proteum/IM to avoid compiling every mutant in a separate file.

Considering $\mathcal{MuDeL}$ as a language to define mutants, any of these approaches can be used, although the most natural one is the individual compilation schema. For instance, a specialized $\mathcal{MuDeL}$ compiler can be constructed, which will generate one or more mutant schematas instead of individual mutants. However, it is necessary to take into account the semantics of the target language, since the way several mutants can vary from one language to another. This is an interesting point for future research. Nonetheless, it is important to highlight that a language like $\mathcal{MuDeL}$ can help in this context, providing a uniform notation and precise semantics for describing the construction of each mutant.

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, because 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 and validation of the mutant operators before delivering a mutant environment is very costly and time-consuming.

In this paper we presented the $\mathcal{MuDeL}$ language as a device for describing mutant operators and generating a mutant generator prototyping module. The language is based on the transformational paradigm and also uses some concepts from logical programming. Being defined in $\mathcal{MuDeL}$, an operator can be "compiled" and the respective mutants can be generated using the `mudelgen` system. $\mathcal{MuDeL}$ and `mudelgen` together form a powerful mechanism to develop mutant operators. The mutant operators can be validated either formally (with the facilities of DS Oracle) or manually (with the facilities of $\mathcal{MuDeL}$ Animator).

The design decisions we have taken lead us one step further towards the achievement of our goals. There are some points that need to be further investigated. For instance, $\mathcal{MuDeL}$ was mainly designed 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. For example, some program mutant operators might need knowledge that are not readily available, such as the set of variables defined prior to a specific point in the program, or whether a method overrides the method of a parent class. Although, these situations can be dealt with `assert` and `consult` operations, we realize that these mutant operators are not easy to write nor are the definition easy to follow. Indeed, the problem of dealing with context aware transformation is a hard problem for any transformational language [33]. We are still investigating how these situations can be more suitably handled. For instance, we observe some idioms in the mutant operators that might be candidates to be included in the language as primary operations.

The experiments we have carried out with $\mathcal{MuDeL}$ involved languages for which there were supporting tools, namely Petri nets and C programs. Although fully useful in demonstrating its potential usage, these experiments are not a complete validation. Right now, we are working on a project where Java mutant operators are being described, what will further contribute towards the validation of the ideas presented herein.

There are tasks that are hard, cumbersome or even impossible to be carried out only with the construction $\mathcal{MuDeL}$ embodies. As example, we can cite arithmetics and string manipulation. To tackle this problem, we are currently developing an API (application programming interface) to allow the implementation and inclusion of *built-in* rules written in a conventional programming language, namely, in C++. We, then, keep the kernel of $\mathcal{MuDeL}$ tiny, whereas built-in rules can be provided for any further need we have to take care of.

Some forthcoming steps in this research include:

- To develop an integrated development environment (IDE), providing features to edit the context-free grammar, the mutant operator and the original product in a manner as uniform as possible, and also providing features to compile, execute and debug the mutant operator definition `mudelgen` is currently operated by means of command-line invocations and has some limited graphical interaction (with $\mathscr{MuDeL}$ 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.
- To integrate the $\mathscr{MuDeL}$ 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 $\mathscr{MuDeL}$, i.e., a tool with multi-language support.
- To investigate the relationship between syntax and semantic mutation.

References

[1] Budd AT. Mutation analysis: ideas, examples, problems and prospects. Computer program testing. Amsterdam: North-Holland Publishing Company; 1981. p. 129–148.
[2] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. IEEE Computer 1978;11(4):34–41.
[3] Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation testing applied to validate specifications based on statecharts. In: ISSRE—international symposium on software reliability systems. 1999. p. 210–9.
[4] Fabbri SCPF, Maldonado JC, Masiero PC, Delamaro ME. Mutation analysis for the validation of Petri net based specifications. In: 8th workshop of software quality, Curitiba, PR, 1994 [in Portuguese].
[5] Souza SRS, Maldonado JC, Fabbri SCPF, Lopes de Souza W. Mutation testing applied to Estelle specifications. Quality Software Journal 2000;8(4):285–301 [publicado também no 33rd Hawaii Internacional Conference on System Sciences, 2000].
[6] Sugeta T, Maldonado JC, Wong WE. Mutation testing applied to validate SDL specifications. In: Hierons RM, Groz R, editors, 16th IFIP international conference on testing of communicating systems—TestCom2004, Oxford, UK, 2004.
[7] Kovács G, Pap Z, Viet DL, Wu-Hen-Chang A, Csopaki G. Applying mutation analysis to SDL specifications. In: Reed R, Reed J, editors, SDL 2003: system design—11th SDL forum. Lecture notes on computer science, vol. 2708, Springer, Heidelberg, Stuttgart, Germany, 2003. p. 269–84.
[8] Probert RL, Guo F. Mutation testing of protocols: principles and preliminary experimental results. In: Proceedings of the IFIP TC6 third international workshop on protocol test systems. Amsterdam: North-Holland, 1991. p. 57–76.
[9] Ritchey RW. Mutating network models to generate network security test cases. In: Mutation 2000, San Jose, California, 2000. p. 101–8.
[10] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? In: International conference on software engineering. St. Louis, MO, USA, 2005. p. 402–11.
[11] Offutt AJ, Rothermel G, Zapf C. An experimental evaluation of selective mutation. In: 15th international conference on software engineering. Baltimore, MD: IEEE Computer Society Press; 1993. p. 100–7.
[12] Wah KSHT. A theoretical study of fault coupling. Software Testing, Verification and Reliability 2000;10(1):3–45.
[13] Nakagawa EY, Maldonado JC. Software-fault injection based on mutant operators. In: Anais do XI Simposio Brasileiro de Tolerancia a Falhas, Florianopolis, SC, 2001. p. 85–98.
[14] Agrawal H. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center/Purdue University; March 1989.
[15] DeMillo RA, Guindi DS, King KN, McCracken WM, Offutt AJ. An extended overview of the Mothra testing environment. In: Second workshop on software testing, verification and analysis, Baniff, Canadá, 1988.
[16] Kim S, Clark JA, Mcdermid JA. The rigorous generation of Java mutation operators using HAZOP. In: 12th international conference on software & systems engineering and their applications (ICSSEA'99), 1999.
[17] Ma Y-S, Kwon Y-R, Offutt J. Inter-class mutation operators for Java. In: 13th international symposium on software reliability engineering—ISSRE'2002, Annapolis, MD, 2002.
[18] Delamaro ME, Maldonado JC, Mathur AP. Interface mutation: an approach for integration testing. IEEE Transactions on Software Engineering 2001;27(3): 228–47.
[19] Alexander RT, Bieman JM, Ghosh S, Ji B. Mutation of Java objects. In: 13th International symposium on software reliability engineering, 2002. p. 341–51.
[20] Fabbri SCPF, Maldonado JC, Delamaro ME, Masiero PC. 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, Talca, Chile, 1999. p. 96–104.
[21] Vincenzi AMR, Nakagawa EY, Maldonado JC, Delamaro ME, Romero RAF. Bayesian-learning based guidelines to determine equivalent mutants. International Journal of Software Engineering and Knowledge Engineering 2002;12(6):675–90.
[22] Vincenzi AMR, Simão AS, Delamaro ME, Maldonado JC. Muta-pro: towards the definition of a mutation testing process. Journal of Brazilian Computer Society 2006;12(2):47–61.
[23] Neighbors J. The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering 1984;10(5):564–74.
[24] Bratko I. Prolog programming for artificial intelligence. 2nd ed., Wokingham, England, Reading, MA: Addison-Wesley; 1990.
[25] Allison L. A practical introduction to denotational semantics. Cambridge, UK: Cambridge University Press; 1986.
[26] Stoy JE. Denotational semantics: the Scott–Strachey approach to programming language theory. Cambridge, MA: MIT Press; 1977.
[27] Simão AS, Maldonado JC, Bigonha RS. Using denotational semantics in the validation of the compiler for a mutation-oriented language. In: Proceedings of 5th workshop of formal methods, Gramado, RS, 2002. p. 4–19.
[28] Weyuker EJ. On testing non-testable programs. Computer Journal 1982;25(4):465–70.
[29] Delamaro ME, Maldonado JC, Mathur AP. Proteum: a tool for the assessment of test adequacy for C programs: user's guide. Technical Report SERC-TR-168-P, Software Engineering Research Center, Purdue University, West Lafayette, IN; April 1996.
[30] Fabbri SCPF. Mutation analysis in the context of reactive systems: a contribution to establishing testing and validation strategies. PhD thesis, IFSC/USP, São Carlos, SP; 1996 [in Portuguese].
[31] McDermid JA, Pumfrey DJ. A development of hazard analysis to aid software design. In: Compass'94: 9th annual conference on computer assurance. Gaithersburg, MD: National Institute of Standards and Technology; 1994. p. 17–26.
[32] Cordy JR, Carmichael IH, Halliday R. The TXL programming language—version 8. Technical Report, Department of Computing and Information Science, Queen's University, Kingston, Canada; April 1995.
[33] Cordy JR, Dean T, Malton A, Schneider K. Source transformation in software engineering using the txl transformation system. Technical Report 13; 2002.
[34] Salomaa A. Formal languages. New York: Academic Press; 1973.
[35] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques and tools. Reading, MA: Addison-Wesley; 1985.
[36] Simão AS, Ambrosio AM, Fabbri SCPF, Amaral AS, Martins E, Maldonado JC. Plavis/FSM: an environment to integrate FSM-based testing tools. In: Caderno de Ferramentas do XIX Simposio Brasileiro de Engenharia de Software, Uberlandia, MG, 2005.

[37] Simão AS. Mutation analysis application in the context of testing and validation of coloured petri nets. PhD thesis, ICMC/USP, São Carlos, SP; 2004.
[38] Fabbri SCPF. 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; 1996.
[39] Yano T, Simão AS, Maldonado JC. Estudo do teste de mutação para a linguagem standard ML. In: Solar M, Fernandez-Baca D, Cuadros-Vargas E, editors, 30ma Conferencia Latinoamericana de Informatica (CLEI2004). Sociedad Peruana de Computacion, 2004. p. 734–44, iSBN 9972-9876-2-0.
[40] Kim S, Clark JA, Mcdermid JA. Class mutation: mutation testing for object-oriented programs. In: Object-oriented software systems, Net.ObjectDays'2000, Erfurt, Germany, 2000.
[41] Kim S, Clark JA, Mcdermid JA. Object-oriented testing strategies using the mutation testing. Software Testing Verification and Reliability 2001;11:207–25.
[42] Mason T, Brown D. Lex & Yacc. O'Reilly; 1990.
[43] Hansen MR, Rischel H. Introduction to programming using SML. Reading, MA: Addison-Wesley; 1999.
[44] Wong E, Maldonado JC, Delamaro ME, Mathur AP. Constrained mutation for C programs. In: 8° Simposio Brasileiro de Engenharia de Software, Curitiba, Brasil, 1994. p. 439–52.
[45] Untch R, Harrold MJ, Offutt J. Mutation analysis using mutant schemata. In: International symposium on software testing and analysis, Cambridge, MA, 1993. p. 139–48.