Using Denotational Semantics in the Validation of the Compiler for a Mutation-Oriented Language

Adenilso da Silva Simão[1], José Carlos Maldonado[1], and Roberto da Silva Bigonha[2]

[1] Departamento de Ciências de Computação e Estatística
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo – Campus de São Carlos
Av. Trabalhador Sãocarlense, 400
Cx. Postal 668
CEP. 13560-970
São Carlos — São Paulo — Brazil
{adenilso,jcmaldon}@icmc.usp.br

[2] Departamento de Ciências de Computação
Instituto de Ciências Exatas
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627
CEP. 31270-010
Belo Horizonte — Minas Gerais — Brazil
{bigonha}@dcc.ufmg.br

Abstract. Systems which produce large amount of output are very difficult to teste due to the expense of checking whether the outputs are correct or not. In other words, there is no simple oracle that can establish the acceptance of the outputs. For such systems, one plausible alternative is to use a second implementation as an *automated oracle*. The automated oracle is then run with the test cases applied to the system under testing and the outputs are compared.
$\mathcal{MuDeL}$ is a language for describing mutant operators and is supported by `mudelgen`, a system that generates the actual mutants from a given description. Usually, the number of mutants that are generated is very large. Therefore, testing `mudelgen` is very difficult. To tackle this problem, we have described the semantics of $\mathcal{MuDeL}$ using *denotational semantics* and developed an automated oracle by implementing this semantics in the SML language. In this paper, we describe the results of this approach. We indicate how using a formal framework (i.e., denotational semantics) and a functional language (i.e., SML) helps improving the confidence of the `mudelgen` system.

Keywords: Denotational Semantics, Functional Programming, SML, Mutation Testing, Testing

1 Introduction

The applicability and feasibility of testing rely heavily on the existence and tractability of the oracle problem and the availability of supporting tools. An oracle is a mechanism that is able to determine whether the output obtained by executing the system under test with a given input is correct or not. Although the existence of an oracle for a given system is usually simply assumed [14, 15], there are some classes

of systems for which no such a mechanism exists. As pointed out by Weyuker [14], these systems can be regarded as *non-testable*. Examples of non-testable systems are those for which the amount of output generated is prohibitively large for being exhaustively examined. Even if an oracle potentially exists, it is very costly to determine the correctness of the resulting output.

An important and desirable feature of an oracle mechanism is that the decision of acceptance must be taken on consistent grounds. This emphasizes the importance of the system specification. The specification should define the expected behavior of the system. It can range from informal (e.g.natural language) to formal approaches (e.g.denotational semantics), passing through semi-formal ones (e.g.UML). In some cases, formal methods are required, since they allow rigorous verification of the results, as well as provide a basis for automating the oracle.

$\mathcal{MUDEL}$ [11] is a transformational-based language developed for describing mutant operators [4, 5, 8], which are key components in mutation testing [3, 6] — a fault-based testing criterion. Assuming a fixed grammar, a mutant operator description in $\mathcal{MUDEL}$ can be thought of as a tree manipulator. A source program is converted into a syntax tree. After execution, a set of trees is produced. The actual mutants are obtained by traversing these trees. We developed a system — `mudelgen` — for compiling a mutant operator description and actually generating the mutants. The number of mutants generated is often very large and manually checking them is very costly and error-prone. This fact severely hinders the tractability of the oracle. Therefore, testing `mudelgen` is a hard task, due mainly to the amount of output yielded, and we can consider it a non-testable system. To cope with this problem, we adopted an approach that can be summarized in two steps. Firstly, we employed *denotational semantics* [1, 12] to formally define the semantics of the $\mathcal{MUDEL}$ language. Secondly, supported by the fact that denotational semantics is primarily based on lambda calculus, we used the language SML [7], which is also based on lambda calculus, to code and run the denotational semantics of $\mathcal{MUDEL}$. We included the SML implementation into `mudelgen` and introduce a new executing mode: the *testing* mode. In the *testing* mode, both the actual and SML implementations are executed and the mutants are compared. Any discrepancy is reported. With this approach, we can automatically derive the expected output and provide an automated oracle.

This paper is organized as follows. Basic concepts needed for the discussion in the remaining sections are presented in Section 2. In Section 3 we introduce the $\mathcal{MUDEL}$ language and informally present its semantics. The formal, denotational semantics of the $\mathcal{MUDEL}$ is presented in Section 4. In Section 5 we illustrate how this semantics can be mapped onto SML code, allowing its execution. The overall execution schema of the oracle is also presented. Finally, in Section 6 we make a general discussion on the strengths and limitations of our approach and point some future work.

2 Basic Concepts

2.1 Mutation Testing and Mutant Operators

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. The

main idea behind Mutation Testing is to employ a set of alternative products (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* [8]. 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 Mutation 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 tool design and construction 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 the implementation of supporting tools. One of the primary motivations of the $\mathcal{MuDeL}$ language (and `mudelgen` thereby) is to provide such a mechanism.

2.2 Grammars and Syntax Trees

In this section we present a brief introduction of grammar and language theories, needed for the discussion that follows. A thorough presentation can be found elsewhere [10]. Syntax grammars are finite devices to describe usually infinite languages. Given a grammar G, we have $L(G)$ as the set of all sentences that can generate by the productions in G. Most, if not all, programming or computer language is characterized by a grammar. 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 [13]. 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 4-tuple $G = (N, T, S, R)$, where N is the set of non-terminal symbols, T is the of terminal symbols, $S \in N$ is a non-terminal symbol referred to as initial symbol, and R is the production rules (with $R \subseteq N \times (N \cup T)^*$). Rather informally, a production rule of the form (n, α) states that the non-terminal symbol n (the *left-hand* symbol) can be replaced by the sequence α (the *right-hand* symbol sequence) of terminal and non-terminal symbols without "inflicting" the grammar. From a sequence $\gamma\langle n\rangle\delta$, we can derive another sequence of form $\gamma\alpha\delta$, for any production (n, α). 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 of 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 the children nodes with labels α_1, α_2, ..., α_k, then there exists a production of form $\langle n \rangle ::= \alpha_1 \alpha_2 \ldots \alpha_k$. If by collecting the terminal symbols in a depth-first, left-to-right traversal of a syntax tree t of grammar G we obtain a sequence φ, then t is a syntax tree of φ w.r.t. G.

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. As a way to distinguish from ordinary identifiers, we prefix the meta-variables with a colon ':'. In the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language, 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 $\langle A \rangle$. In a more elaborated pattern definition, 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, [S< (:a) * :b>] is a pattern (for a given grammar) whose type is S and which includes two meta-variables, namely, '`:a`' and '`:b`'. Note that inside the angle brackets the grammar of the product, rather than the $\mathcal{M}u\mathcal{D}e\mathcal{L}$'s grammar, is to be respected.

There are two main operations in the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language that involve syntax and pattern trees: *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 (i.e., bound) either to a closed pattern tree or to another meta-variable, in a way that makes them unrestrictedly interchangeable. In case of failure, no meta-variable unification occurs. 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 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.

3 Informal Introduction to $\mathcal{M}u\mathcal{D}e\mathcal{L}$

The $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language is concerned with describing how the syntax tree of the original program is to be changed in order to generate the syntax trees of the mutants. This is basically accomplished by composing matching and replacing operations. Each basic operation has a usual semantics, which can be tuned by applying some modifiers. There are also some meta-operations that can be used to control how and whether the mutants are generated. Therefore, the fundamental elements of $\mathcal{M}u\mathcal{D}e\mathcal{L}$ are basic operations, modifiers, combinators and meta-operations. In

the next sections we provide a brief description of these elements, needed for the discussion in Section 4. A more thorough description can be found in [11].

3.1 Basic Operations

There are three basic operations in the language. In each operation, there is an implicit context tree. How this context tree is used depends on the operation in consideration. An operation can either succeed or fail. If an operation fails, the mutant in question is not generated.

matching operation A matching operation "**match** m" demands the pattern m be matched (i.e., unified, borrowing the term from the language Prolog [2]) against the context tree. It succeeds if a unification exists, and fail otherwise.

replacing operation A replacing operation "**replace** r **by** b" demands the pattern r be matched against the context tree and, if a unification exists, the context tree is replaced by the pattern b.

grouping operation The grouping operation[1] "((op))", where op is an operation, has the purpose of grouping the scope of modifiers, overriding precedences of combinators and delimiting the scope of a **cut** operation, explained later.

3.2 Modifiers

There are three modifiers we can apply to a basic operation. Each of them can be applied (prefixed) at most once to the same operation.

context definition The context definition ":m @@ op" is used to define the tree (if any) bound to the meta-variable :m as the context of the modified operation op.

negation The negation modifier "~ op" is used to invert (negate) the result of the modified operation. That is, if op succeeds, the negated operation ~ op fails. Otherwise, if op fails, the negated operation ~ op succeeds with no unification.

in depth The in depth modifier "∗ op" is used to indicate that the modified operation op is to be applied not only to the context tree, but also each of its sub-trees, recursively. This implies that in this point several alternatives of executions are possible. In the execution semantics of $\mathcal{MuDeL}$, every possible alternative will be tried.

3.3 Combinators

There are two combinators we can employ to compose complex operations from the basic ones.

sequence combinator The sequence combinator "op1 ;; op2" introduces a list of operations to be applied in sequence. The complex operation succeeds if every operation succeeds in sequence.

alternative combinator The alternative combinator "op1 || op2" introduces a list of alternative operations that will be exclusively and exhaustively tried in turn.

The sequence combinator has a higher precedence than has the alternative combinator. Therefore, "a ;; b || c ;; d" is interpreted as a list of alternative operations composed of two (compound) operations, namely, "a ;; b" and "c ;; d". The grouping can be used to override the precedence, as in "a ;;((b || c));; d".

[1] This is not a proper basic operation. We classified it as such because, as the other basic operations, the it can also have modifiers.

3.4 Meta-Operations

There are three meta-operations that improve the control over the execution of a mutant operator.

donothing The donothing operation always succeeds.

abort The abort operation always fails.

cut The cut operation was designed after the "cut" predicate of Prolog [2]. A cut operation succeeds but has the side effect of "cutting" any alternative in the innermost grouping. In other words, after an always succeeding execution of a cut operation, any alternative in the innermost grouping is forgotten.

3.5 An Example

In this section we present an example to exemplify the main concepts of the $\mathcal{MuDeL}$ language. In Figure 1 we present a simple mutant operator that is meant to change every *while* statement into a *do-while* and[2] also change the control expression into 0 and 1, if it does not already equal to 0 and 1, respectively. Figure 2(a) presents a simple C program and Figures 2(b)-(e) present the mutants that will be generated for this program w.r.t. the operator in Figure 1.

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

Fig. 1. A multipurpose *while* mutant operator.

The replacing operation in lines 1 and 2 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 4 through 16 makes changes in this control expression. Observe that the context pattern declaration in line 4 affects the whole group, and, consequently, every operation therein.

The (negated) matching in line 5 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 7 and 8, 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 15 is tried and a mutant is generated only with the replacement of line 1.

[2] Indeed, different mutant operators would be better than a general one like this. We do so to illustrate $\mathcal{MuDeL}$'s features.

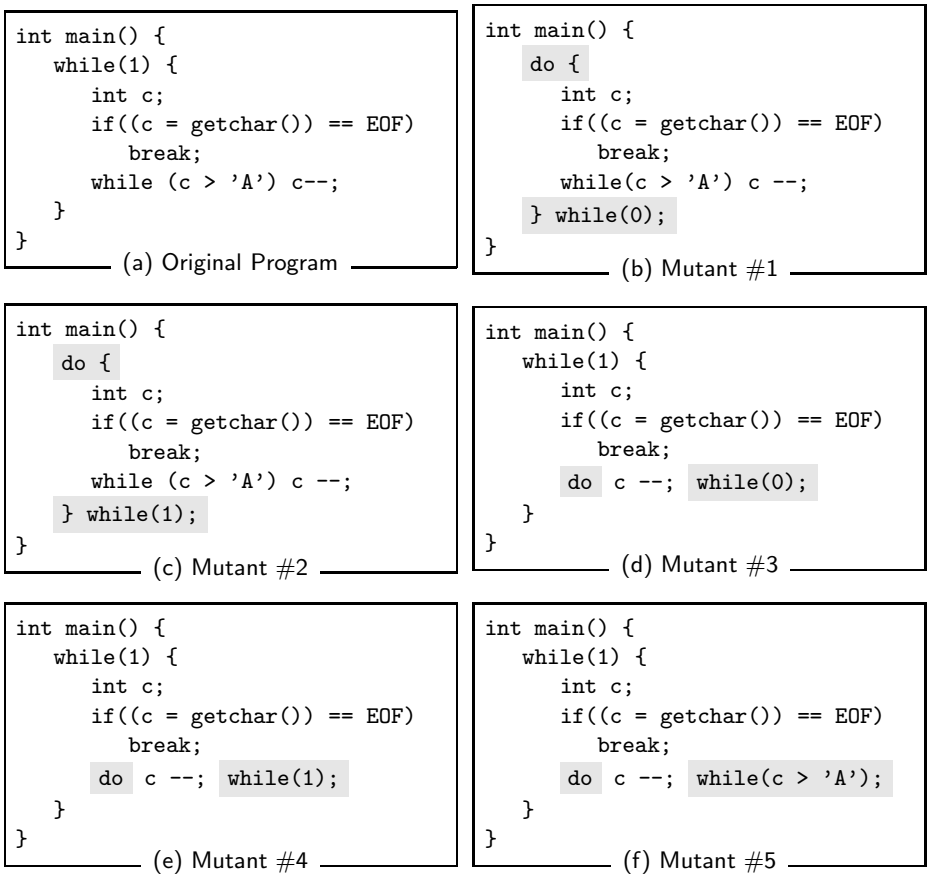

Fig. 2. (a) Original Program. (b)-(e) Mutants generated by operator in Figure 1.The mutated parts of the code are highlighted.

Analyzing how the mutants are generated in this example illustrates the way $\mathcal{MuDeL}$ processes a mutant operator description. The replacing operation (lines 1 and 2) 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 4 through 16 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 of a list of three choices: the first choice is in lines 5 through 8, the second one is in lines 10 through 13, and the last one is in line 15. 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 2(b)-(c) are generated by replacing the outermost while of the program in Figure 2(a) and applying the first and the third choices, respectively. (Observe that the second choice does not generate a mutant, since the operation in line 10 does not succeed.) On the other hand, Mutants #3, #4 and #5 in Figures 2(d)-(f) are generated by replacing the innermost while and applying each of the choices, respectively.

4 Formal Definition

4.1 Domains

This section describes the semantic domains used to specify the denotational semantics of the $\mathcal{MuDeL}$ language.

Basic Domains

The domain of locations Loc is a totally ordered, infinite domain. Each location is associated with a tree node. One possibility to define Loc is to use the natural numbers. Therefore

$$\mathsf{Loc} = \{0, 1, ...\}$$

The domain of non-terminals $\mathsf{NonTerm}$ is the finite domain whose elements are in one-to-one correspondence to the non-terminal symbols of the grammar in consideration. Analogously, the domain of terminals Term are the finite domain whose elements is in one-to-one correspondence to the terminal symbols of the grammar.

The the elements of domain Rec are composed of a sequence of locations and a non-terminal and represents an inner node in the syntax tree.

$$\mathsf{Rec} = \mathsf{Loc}^* \times \mathsf{NonTerm}$$

The domain Free represents the value of an unbound meta-variable and possesses a single element **free**.

$$\mathsf{Free} = \{\mathbf{free}\} \times \mathsf{NonTerm}$$

The domain Undef possesses the single element **undef**, for a non-allocated location.

$$\mathsf{Undef} = \{\mathbf{undef}\}$$

The state that can occur during the execution of a mutant operator basically consists of a particular configuration of the syntax tree and the pattern trees, as well as in the assignment of meta-variables. The meta-variables are associated with locations, in such a way that we can formally represent the states as a function from locations to either an undefined value, a terminal (which represents a leaf node in a tree), another location, a free value (which represents an unbound meta-variable) or a record of locations (which represent a non-terminal and its children nodes).

$$\mathsf{State} = \mathsf{Loc} \to (\mathsf{Undef} + \mathsf{Term} + \mathsf{Loc} + \mathsf{Free} + \mathsf{Rec})$$

Note that both the unbound meta-variable and the record of children nodes are marked with a non-terminal, which is its type. With this modeling, we can represent any syntax tree or pattern tree. (Indeed, we can represent a graph with an arbitrary topology.) One tree can be identified by indicating its root location.

A mutant is an alternative program and, therefore, consists of a list of terminal symbols. Thus

$$\mathsf{Mut} = \mathsf{Term}^*$$

The result of executing a mutant operator is defined as a list of mutants, as follows:

$$\mathsf{Ans} = \mathsf{Mut}^*$$

Continuation Domains

In the terminology of denotational semantics theory, a continuation is a function that represents the effect of further executing the program. In the $\mathcal{M}u\mathcal{D}e\mathcal{L}$ language, there are two kinds of continuations: alternative continuation and normal continuation.

An *alternative continuation* is a mapping from a partial answer with the mutants generated so far to another (possibly partial) answer. An alternative continuation represents a point where the execution can continue on request, either because a mutant is generated and another will be tried, or because the current execution fails. Thus

$$\mathsf{Ka} = \mathsf{Ans} \rightarrow \mathsf{Ans}$$

A *normal continuation* represents the normal execution until the end of a mutant operator. It includes (i) a state (the current state), (ii) an alternative continuation (to be tried on fail), (iii) an alternative continuation (that represent where to continue if a **cut** operation is executed), and (iv) a partial answer (the mutant generated so far) and generates another answer. Therefore

$$\mathsf{Kn} = \mathsf{State} \rightarrow \overbrace{\mathsf{Ka}}^{retry} \rightarrow \overbrace{\mathsf{Ka}}^{cut} \rightarrow \mathsf{Ans} \rightarrow \mathsf{Ans}$$

or

$$\mathsf{Kn} = \mathsf{State} \rightarrow \mathsf{Ka} \rightarrow \mathsf{Ka} \rightarrow \mathsf{Ka}$$

4.2 Formal Syntax

The syntax of the language is presented here to give a framework in which to specify the denotational semantics. In this paper, for sake of space, we assume that the syntactical analysis was already made, in such a way that syntax and pattern trees can be handled just by identifying its root node. Moreover, we assume that every meta-variable is replaced by a respective free location. Therefore, the primary syntactic elements of a mutant operator are trees and operations. Figure 3 presents the abstract syntax of a mutant operator.

Op ::= **match** m
| **replace** r **by** b
| ((**Op**$_1$))
| :m **@@** **Op**$_1$
| ˜**Op**$_1$
| $*$ **Op**$_1$
| **Op**$_1$;; **Op**$_2$
| **Op**$_1$ || **Op**$_2$
| **donothing**
| **abort**
| **cut**

Fig. 3. Abstract Syntax of $\mathcal{M}u\mathcal{D}e\mathcal{L}$ Language.

4.3 Semantic Valuations

The semantic valuation function $\mathcal{A}$ assigns the semantics to each operation. In order to define the resulting answer, this function should be provided with (i) the location (which represents the context tree), (ii) the *normal* continuation (which represents the next operation to execute), (iii) the current state, (iv) the *retry* continuation (which is an alternative continuation and represents the next point to be tried), (v) the *cut* continuation (which is an alternative continuation and represents what will be the retry continuation when a **cut** operation is executed), and (vi) a partial answer (with the mutants generated so far). Therefore

$$\mathcal{A} : \mathbf{Op} \to \mathsf{Loc} \to \mathsf{Kn} \to \mathsf{State} \to \mathsf{Ka} \to \mathsf{Ka} \to \mathsf{Ans} \to \mathsf{Ans}$$

or

$$\mathcal{A} : \mathbf{Op} \to \mathsf{Loc} \to \mathsf{Kn} \to \mathsf{Kn}$$

The match operation simply consists of the *unify* function. Note that the *unify* function (defined later is this section) takes two locations representing the root nodes of the tree to be matched and changes the state in order to reflect the effects of any potential free location unification.

$$\mathcal{A}[\![\mathbf{match}\ \mathsf{m}]\!]\ l = \mathit{unify}\ \ l\ \mathsf{m}$$

The replace operation consists of matching the context and pattern locations and continuing on success in the updated state[3].

$$\mathcal{A}[\![\mathbf{replace}\ \mathsf{r}\ \mathbf{by}\ \mathsf{b}]\!]\ l\ k_n\ s = \mathit{unify}\ \mathsf{r}\ l\ \overbrace{(k_n\ s[l/\mathsf{b}])}^{\textit{normal}\ \text{cont.}}\ \ s$$

The semantics of the grouping operation is the semantics of the inner operation with the necessary adjustments to limit the effect scope of a **cut** operation. Observe that the *cut* continuation in the *normal* continuation of the inner operation is fixed (by the lambda function).

$$\mathcal{A}[\![((\ \mathsf{op}\ ))]\!]\ l\ k_n\ s\ k_r\ k_c = \mathcal{A}[\![\mathsf{op}]\!]\ l\ \overbrace{(\lambda\ s'\ k'_r\ k'_c.\ k_n\ s'\ k'_r\ k_c)}^{\textit{normal}\ \text{cont.}}\ \ s\ k_r\ k_r$$

The semantics of the context definition modifier is the semantics of the modified operation on the location associated with the meta-variable.

$$\mathcal{A}[\![\mathsf{:m}\ @@\ \mathsf{op}]\!]\ l = \mathcal{A}[\![\mathsf{op}]\!]\ (\mathit{loc}\ \mathsf{:m})$$

where $\mathit{loc}\ \mathsf{:m}$ is the location associated with the meta-variable :m. (Note that, we assume the mutant operator was preprocessed, so that a location can unambiguously represent a meta-variable.)

The negation modifier inverts the continuations of the modified operation; the *retry* continuation is used as the *normal* continuation, and vice versa.

$$\mathcal{A}[\![\tilde{}\ \mathsf{op}]\!]\ l\ k_n\ s\ k_r\ k_c = \mathcal{A}[\![\mathsf{op}]\!]\ l\ \overbrace{(\lambda\ s'\ k'_r\ k'_c.\ k_r)}^{\textit{normal}\ \text{cont.}}\ \ s\ \overbrace{(k_n\ s\ k_r\ k_c)}^{\textit{retry}\ \text{cont.}}\ \ k_r$$

In spite of its conceptual simplicity, the *in depth* modifier has a complex semantic valuation. We employ an auxiliary semantic valuation function $\mathcal{G}$ (defined later in this section), that recursively applies the modified operation to every sub-tree of the

[3] The mapping $s[l/\mathsf{b}]$ is identical to s, except that l is mapped onto b.

context tree. Therefore, the function $\mathcal{G}$ will generate several alternative continuations.

$$\mathcal{A}[\![* \ \mathsf{op}]\!]\ l\ k_n\ s\ k_r\ k_c = \mathcal{A}[\![\mathsf{op}]\!]\ l\ k_n\ s\ \overbrace{(\mathcal{G}[\![\mathsf{op}]\!]\ l\ k_n\ s\ k_r\ k_c)}^{retry\ \text{cont.}}\ k_c$$

The semantics of a complex operation composed with the sequence combinator is the semantics of the first operation using the second one as the *normal* continuation.

$$\mathcal{A}[\![\mathsf{op1}\ ;;\ \mathsf{op2}]\!]\ l\ k_n = \mathcal{A}[\![\mathsf{op1}]\!]\ l\ \overbrace{(\mathcal{A}[\![\mathsf{op2}]\!]\ l\ k_n)}^{normal\ \text{cont.}}$$

The semantics of a complex operation composed with the *alternative* combinator is the semantics of the first operation using the second one as the *retry* continuation.

$$\mathcal{A}[\![\mathsf{op1}\ ||\ \mathsf{op2}]\!]\ l\ k_n\ s\ k_r\ k_c = \mathcal{A}[\![\mathsf{op1}]\!]\ l\ k_n\ s\ \overbrace{(\mathcal{A}[\![\mathsf{op2}]\!]\ l\ k_n\ s\ k_r\ k_c)}^{retry\ \text{cont.}}\ k_c$$

The semantics of the **donothing** operation equals to the *normal* continuation.

$$\mathcal{A}[\![\mathbf{donothing}]\!]\ l\ k_n = k_n$$

The semantics of the **abort** operation equals to the *retry* continuation.

$$\mathcal{A}[\![\mathbf{abort}]\!]\ l\ k_n\ s\ k_r\ k_c = k_r$$

The semantics of the **cut** operation equals to the *normal* continuation invoked with the *cut* continuation as the *retry* continuation.

$$\mathcal{A}[\![\mathbf{cut}]\!]\ l\ k_n\ s\ k_r\ k_c = k_n\ s\ k_c\ k_c$$

The semantic valuation function $\mathcal{G}$ possesses the same signature as the function $\mathcal{A}$. It checks with which value the context location is associated in the current state. If the context location is associated with an element of Rec, then the function $\mathcal{G}$ applies the operation to every element of this record, using the auxiliary semantic function $\mathcal{R}$. Otherwise, it tries another alternative (i.e., invokes the *retry* continuation). Thus

$$\mathcal{G}[\![\mathsf{op}]\!]\ l\ k_n\ s\ k_r\ k_c = \mathit{isRec}\ (s\ l) \rightarrow \mathcal{R}[\![\mathsf{op}]\!]\ (\mathit{el}\ 1\ (s\ l))\ k_n\ s\ k_r\ k_c,\ k_r$$

where $\mathit{el}\ n\ d$ is the n-th component of the Cartesian product d [12].

The semantic valuation function $\mathcal{R}$ possesses the same signature as the function $\mathcal{A}$, except that a sequence of locations is employed (instead of a single one). If the sequence of locations is empty, it simply invokes the *retry* continuation. Otherwise, it applies the function $\mathcal{A}$ to the first element and makes the function $\mathcal{R}$ be applied to the remaining ones as an alternative continuation.

$$\mathcal{R}[\![\mathsf{op}]\!]\ r\ k_n\ s\ k_r\ k_c = \mathit{isnull}\ r \rightarrow k_r,$$
$$\mathcal{A}[\![* \ \mathsf{op}]\!]\ (\mathit{hd}\ r)\ k_n\ s\ (\mathcal{R}[\![\mathsf{op}]\!]\ (\mathit{tl}\ r)\ k_n\ s\ k_r\ k_c)\ k_c$$

The semantic valuation function $\mathcal{S}$ defines the semantics of a mutant operator. The semantics of a mutant operator consists of defining the proper function in invoking the semantics of the operation using addmut as the *normal* continuation, print as both *retry* and *cut* continuations and an empty list of mutants as initial answer. Note that the function $\mathcal{S}$ takes as parameters the root node of the syntax tree and the initial state, which are obtained by preprocessing both the source program and the mutant operator.

$$\mathcal{S}[\![\mathsf{op}]\!]\ l\ s = \mathcal{A}[\![\mathsf{op}]\!]\ l\ (\mathit{addmut}\ l)\ s\ \mathit{print}\ \mathit{print}\ \langle\rangle$$

The addmut function takes a location and returns a *normal* continuation that adds the current mutant (obtained by traversing the tree with the location l as root)

to the partial answer and invoking the *retry* continuation in order to try another alternative. Thus

$$\mathit{addmut}\ \ l\ s\ k_r\ k_c\ m = \ k_r((\mathit{traverse}\ \ s\ l) \bullet m)$$

where the function *traverse* is a straightforward depth-first, left-to-right traversal of the tree whose root node is the location l and $\bullet$ is the list concatenation operator.

The *print* function is the identity function and simply concludes the execution with the mutants generated so far.

$$\mathit{print}\ \ m = \ m$$

4.4 Unify

It remains to define the function *unify*. The unification algorithm is similar to, and was designed after, Prolog's one [2, 9]. Its signature is

$$\mathit{unify}\ : \mathsf{Loc} \to \mathsf{Loc} \to \mathsf{Kn} \to \mathsf{State} \to \mathsf{Ka} \to \mathsf{Ka} \to \mathsf{Ans} \to \mathsf{Ans}$$

$$\begin{aligned}
&\mathit{unify}\ \ l_1\ l_2\ k_n\ s\ k_c\ k_r = \\
&\qquad l_1 = \ l_2 \to \ k_n\ s\ k_c\ k_r, && (1)\\
&\qquad \mathit{isUndef}\ (s\ l_1) \vee \mathit{isUndef}\ (s\ l_2) \to \ k_r, && (2)\\
&\qquad \mathit{isTerm}\ (s\ l_1) \vee \mathit{isTerm}\ (s\ l_2) \to \\
&\qquad\qquad ((s\ l_1) = (s\ l_2) \to \ k_n\ s\ k_c\ k_r,\ k_r), && (3)\\
&\qquad \mathit{isLoc}\ (s\ l_1) \to \mathit{unify}\ (s\ l_1)\ l_2\ k_n\ s\ k_c\ k_r, && (4)\\
&\qquad \mathit{isLoc}\ (s\ l_2) \to \mathit{unify}\ \ l_1(s\ l_2)\ k_n\ s\ k_c\ k_r, && (5)\\
&\qquad (\mathit{el}\ 2\ (s\ l_1)) \neq (\mathit{el}\ 2\ (s\ l_2)) \to \ k_r, && (6)\\
&\qquad \mathit{isFree}\ (s\ l_1) \to \ k_n(s[\ l_2/\ l_1])\ k_c\ k_r, && (7)\\
&\qquad \mathit{isFree}\ (s\ l_2) \to \ k_n(s[\ l_1/\ l_2])\ k_c\ k_r, && (8)\\
&\qquad \mathit{isRec}\ (s\ l_1) \wedge \mathit{isRec}\ (s\ l_2) \to \\
&\qquad\qquad \mathit{unifyRec}\ (\mathit{el}\ 1\ (s\ l_1))\ (\mathit{el}\ 1\ (s\ l_2))\ k_n\ s\ k_c\ k_r, && (9)\\
&\qquad k_r && (10)\\
&\mathit{unifyRec}\ \ r_1\ r_2\ k_n\ s\ k_c\ k_r = \\
&\qquad \mathit{isnull}\ \ r_1 \vee \mathit{isnull}\ \ r_2 \to \\
&\qquad\qquad (\mathit{isnull}\ \ r_1 \wedge \mathit{isnull}\ \ r_2 \to \ k_n\ s\ k_c\ k_r,\ k_r), && (11)\\
&\qquad \mathit{unify}\ (\mathit{hd}\ \ r_1)\ (\mathit{hd}\ \ r_2)\ (\mathit{unifyRec}\ (\mathit{tl}\ \ r_1)\ (\mathit{tl}\ \ r_2)\ k_n)\ s\ k_c\ k_r && (12)
\end{aligned}$$

Fig. 4. Definition of the functions *unify* and *unifyRec*.

The definition of *unify* is presented in Figure 4. In order to clarify the meaning of this function, in the following lines we comment each relevant part of its definition. After checking a series of condition, it will invoke either the *retry* or the *normal* continuation. We will refer to these outcomes as **retry** and **continue**, respectively.

(1) If the locations to be matched are the same, then **continue**.
(2) If either location is associated with the **undef** value, then **retry**.
(3) If either location is associated with a terminal, then **continue**, if the values associated to locations are the same, or **retry**, otherwise.
(4) If l_1 is associated with another location, unify this other location with l_2.
(5) Analogous to (4), w.r.t. l_2.
(6) At this point we are sure that the values associated to l_1 and l_2 are either Free or Rec. Whichever the case, each of these values is a pair whose second element

belongs to NonTerm, which is its type. Therefore, if the types of both values are not the same, **retry**.

(7) If l_1 is **free**, then **continue** in a new state in which l_1 is associated l_2.

(8) Analogous to (7), w.r.t. l_2.

(9) If both values belong to Rec, then invoke the auxiliary function *unifyRec* on the respective sequence of locations.

(10) If none of the above cases applies, then **retry**.

The function *unifyRec* will try to unify the two sequences of locations, applying *unify* to their respective elements.

(11) If either sequence is empty, then **continue**, if both are, or **retry**, otherwise.

(12) Otherwise, invoke *unify* for the first elements of both sequences, with as *normal* continuation the application of *unifyRec* to the remaining elements.

5 SML Implementation and its Use as a Validation Mechanism

Since SML [7] is a functional language based on typed lambda calculus, translating from a denotational semantics specification into SML code is quite a straightforward process, apart from some syntax translations. Therefore, we implement the denotational semantics defined in Section 4 in SML. The purpose of this implementation is twofold. Firstly, we could check whether the defined semantics was syntactically correct and whether the domains are properly defined and arranged. Secondly, we were able to "execute" it and, hence, we could employ it as a validation mechanism for the `mudelgen` — a compiler we had implemented for the $\mathcal{MuDeL}$ language.

As a sample of the SML implementation, Figure 5 presents how the semantic valuation functions $\mathcal{A}$, $\mathcal{G}$, $\mathcal{R}$ and $\mathcal{S}$ and the auxiliary function *addmut* and *print* are implemented. Note that the definition of the $\mathcal{A}$ function poses some syntax changes, once it is recursively defined. The most evident aspect is that we can not simplify the definition by eliminating the common, unchanged arguments in the same way we can in the denotational semantics. This occurs because SML demands all clauses have the same explicit signature, and, thus, we had to include all arguments in each one. Nevertheless, by comparing these lines with the respective equation (see Section 4), one can observe similarities between SML and the denotational semantics mentioned previously.

To employ the SML implementation as a validation mechanism, we extend `mudelgen` by including a *testing* mode. Figure 6 depicts the execution of `mudelgen` in testing mode. (The dashed line surrounds the components specific to this mode.) Both source program S and mutant operator Op are inputted and processed by the parse and mutant operator processor (MO processor), respectively. The syntax tree T of the source program and the abstract representation Op' of the mutant operator are obtained. Then T and Op' are processed by both SML and C++ implementations. The outputs are (a) the mutants and (b) a report of any discrepancy between the mutants generated by both implementations. The discrepancy report contains information about differences (if any) in the token streams of every pair of respective mutants (one of each implementation).

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,

```
fun A(mtc m)         l kn s kr kc = match l m kn s kr kc
and A(rpl(r, b))     l kn s kr kc = replace r b l r e kn s kr kc
and A(grp op)        l kn s kr kc = A op l
                     (fn s' => fn kr' => fn kc' => kn s' kr' kc) s kr kr
and A(con(v,op))     l kn s kr kc = A op v kn s kr kc
and A(neg op)        l kn s kr kc = A op l
                     (fn t' => fn kr' => fn kc' => kr) s (kn s kr kc) kr
and A(dep op)        l kn s kr kc = A op l kn s (G op l kn kr kc) kc
and A(sqc(op1,op2)) l kn s kr kc = A op1 l (A op2 l kn) s kr kc
and A(alt(op1,op2)) l kn s kr kc = A op1 l kn s (A op2 l kn s kr kc) kc
and A dnt            l kn s kr kc = kn s kr kc
and A abt            l kn s kr kc = kr
and A cut            l kn s kr kc = kn s kc kc
and G op             l kn s kr kc = case s l of
                          rec(r, t) => R op r kn s kr kc
                        | _         => kr
and R op             r kn s kr kc =
                   case r of
                      []     => kr
                    | (h, t) => A (dep op) h kn s (R op t kn s kr kc) kc
;

fun S op l s = A op l (addmut l) s print print []
and addmut l s kr kc m = kr ((traverse s l)::m)
and print m = m
;
```

Fig. 5. SML code for the semantic valuation functions $\mathcal{A}$, $\mathcal{G}$, $\mathcal{R}$ and $\mathcal{S}$ as well for the auxiliary functions *addmut* and *print*.

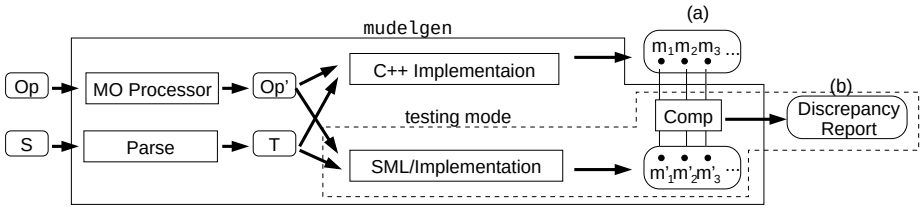

Fig. 6. Oracle Implementation Schema

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.

$\mathcal{MUDEL}$ language is designed for describing mutant operators. The language is based on the transformational paradigm and also uses some concepts from logical programming. The semantics of $\mathcal{MUDEL}$ was defined using denotational semantics, making clear and precise the meaning of every construction in the language. Being described 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 instrument in developing and validating mutant operators.

However, the amount of output produced by executing a mutant operator is usually very large. Therefore, the task of evaluating whether the mutants are generated in accordance with the description is very difficult. This fact characterizes the so-called non-testable programs and hinders the testing of `mudelgen`. In this paper we described how we have employed SML to implement the denotational semantics of $\mathcal{MuDeL}$. We integrated this SML implementation in `mudelgen`, aiming at using it as an automated oracle for the C++ implementation.

The complete SML implementation of the denotational semantics of the $\mathcal{MuDeL}$ language (including the auxiliary functions) takes ±220 lines of code. On the other hand, the implementation of `mudelgen` in C++ has ±5500 lines of code. This total includes the code for parsing and unparsing, as well as the code for interfacing with the SML implementation. We can observe that SML is much more concise, primarily due to the expressiveness power of the continuation based approach [12].

One important question that arises in face of these differences in size and complexity is why to have the C++ implementation at all. We could limit ourselves to implement in C++ only those parts that are not suitable to build in SML (e.g. parsing) and use the denotational semantics based implementation in SML for operator execution. However, a pragmatic answer is the relative inefficiency of SML when compared against C++. Indeed, the SML code runs from 50 to 100 times slower than the C++ code[4].

When a discrepancy in the outputs are identified, we can not know *a priori* which implementation is responsible for the wrong result. We then analyze both and correct them properly. Two errors were found in this way, one in either module. The C++ implementation had a fault in the mechanism that controls the scope of a cut operation. The denotational semantics had a fault in the semantics of the grouping operation. Running the implementation in the testing mode, both errors were exposed.

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 also possesses a fault that makes it produce 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. Whereas we employed an imperative stack-based approach in C++, we extensively used continuation and mappings 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`.

As a forthcoming step in this research, we will investigate how to insert of other checkpoints in the execution of a mutant operator. Currently, only the generated mutants of both implementations are compared. It may be interesting to include other intermediate points in the middle of execution where the internal state can also be compared, increasing the granularity of observation.

[4] For this comparison, we have used the compilers Standard ML of New Jersey, Version 110.0.7 and gcc, Version 2.96, respectively. However, we have not carried out a benchmark in order to thoroughly compare the effieciency of both implementations.

Bibliography

[1] Allison, L. (1986). *A Practical Introduction to Denotational Semantics*. Cambridge University Press, Cambridge, U. K.

[2] Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 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] 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.

[5] DeMillo, R. A. (1988). An extended overview of the Mothra testing environment. In *Second Workshop on Software Testing, Verification and Analysis*, Baniff, Canadá.

[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] Hansen, M. R. and Rischel, H. (1999). *Introduction to Programming using SML*. Addison-Wesley.

[8] 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.

[9] Nicholson, T. and Foo, N. (1989). A denotational semantics for prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665.

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

[11] Simão, A. S. and Maldonado, J. C. (2001). MuDeL: A language and a system for describing and generating mutants. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 240–255, Rio de Janeiro, Brasil.

[12] Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massassuchits.

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

[14] Weyuker, E. J. (1982). On testing non-testable programs. *Computer Journal*, 25(4):465–470.

[15] Zhu, H. (1996). A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255.