`mudelgen`: A Tool for Processing Mutant Operator Descriptions

Adenilso da Silva Simão
Auri Marcelo Rizzo Vincenzi
José Carlos Maldonado
{adenilso,auri,jcmaldon}@icmc.sc.usp.br

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

Abstract

Mutation Testing is a testing approach for assessing the adequacy of a set of test cases by analyzing their ability in distinguishing the product under test from a set of alternative products, the so-called mutants. The mutants are generated from the product under test by applying a set of mutant operators, which systematically yield products with slight syntactical differences. Aiming at automating the generation of mutants, we have designed a language — named $\mathcal{MuDeL}$ — for describing mutant operators. In this paper, we describe the `mudelgen` *system, which was developed to support the language $\mathcal{MuDeL}$.* `mudelgen` *was developed using concepts that come from transformational and logical programming paradigms, as well as from context-free grammar and denotational semantics theories.*

Keywords: *Mutation Testing, Transformational Paradigms, Denotational Semantics, SML.*

1 Introduction

Mutation Testing [4, 6] is a testing approach proposed to assess the quality of a test case suite in revealing some specific classes of faults. In this sense, Mutation Testing can be classified among the fault-based testing techniques. It was originally proposed for program testing [6]. 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 by including some slight syntactical changes that induces specific faults in the product. In the Mutation Testing approach, faults are modeled by *mutant operators* [4]. 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 expressiveness of the set of faults modeled in the mutant operators has great impact in the Mutation Testing cost and effectiveness, and, hence, so do the mutant operators themselves.

In order to be able to precisely define mutant operators and to ease mutant generation, we have designed a language — called $\mathcal{MuDeL}$ (**MU**tant **DE**scription **L**anguage) [11]. The

$\mathcal{MuDeL}$ language captures the underlying concepts that leaded to the mutant operator definition. We have implemented the system `mudelgen` (standing for $\mathcal{MuDeL}$ **Gen**erator), so that a $\mathcal{MuDeL}$ mutant operator description can be "compiled" into an actual mutant operator, enabling the mutant operator designer to validate the definition and, potentially, to improve it. Given a context-free grammar G of a specific language L, the `mudelgen` builds a program $P_{(G)}$ for this language. In its turn, given a mutant operator description and the original product, $P_{(G)}$ compiles the description and generates the mutants. Both $\mathcal{MuDeL}$ and `mudelgen` are designed with concepts from transformational [9] and logic [3] paradigms, as well as from denotational semantics theory.

This paper is organized as follows. In Section 2 we present the main features of `mudelgen`. The overall structure of `mudelgen` and the most important implementation aspects are presented in Section 3. In Section 4 we discuss how the denotational semantics formalism was used in the validation process of `mudelgen`. In Section 5 we illustrate a visual tool which allows the inspection of a mutant operator execution, including some limited debugging capabilities. Finally, we present some concluding remarks in Section 6.

2 Main Features

Mutation Testing can be, and indeed has been, applied to several different contexts and languages, ranging from imperative programming languages (e.g. C [5]) to formal specification techniques (e.g. Petri Nets [12]). Therefore, a mechanism to describe mutant operators should ideally be able to deal with all, or at least most, of those applications. A common element upon which we can construct a generic approach is the grammar description of their languages. Indeed, each of these languages can be characterized by a grammar. In particular, we are concerned with languages which can be described by context-free grammars [10]. $\mathcal{MuDeL}$ can, thus, be instantiated to a particular grammar; this means that the validity of a mutant description can only be determined by considering a particular grammar.

A $\mathcal{MuDeL}$ description of a mutant operator is basically a set of matching and replacing operations that are combined in the proper way in order to indicate how portions of the original product are to be changed [11]. Both matching and replacing operations specify some patterns which are searched for in the syntax tree. These patterns can include meta-variables. A meta-variable represents a placeholder for a specific kind of syntax non-terminal symbol.

3 Implementation Aspects

To develop `mudelgen`, we consider context-free grammars as input data. We have chosen to employ compiler development tools to manipulate these grammars. We use `bison` and `flex`, which are open source programs similar to, respectively, `yacc` and `lex` [8]. 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 [1, 8, 10]. 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 [8]. 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 [8]. Indeed, these files can be thought of as minimal standard `yacc` and `lex` inputs, from which all so-called semantic actions were stripped off.

Although `mudelgen` can be regarded as a unique system, it is actually composed by 3 modules, which are executable programs: `treegen`, `opdescgen` and `linker`. Figure 1 depicts how these modules are related to each other. It also illustrates the overall execution schema of `mudelgen`.

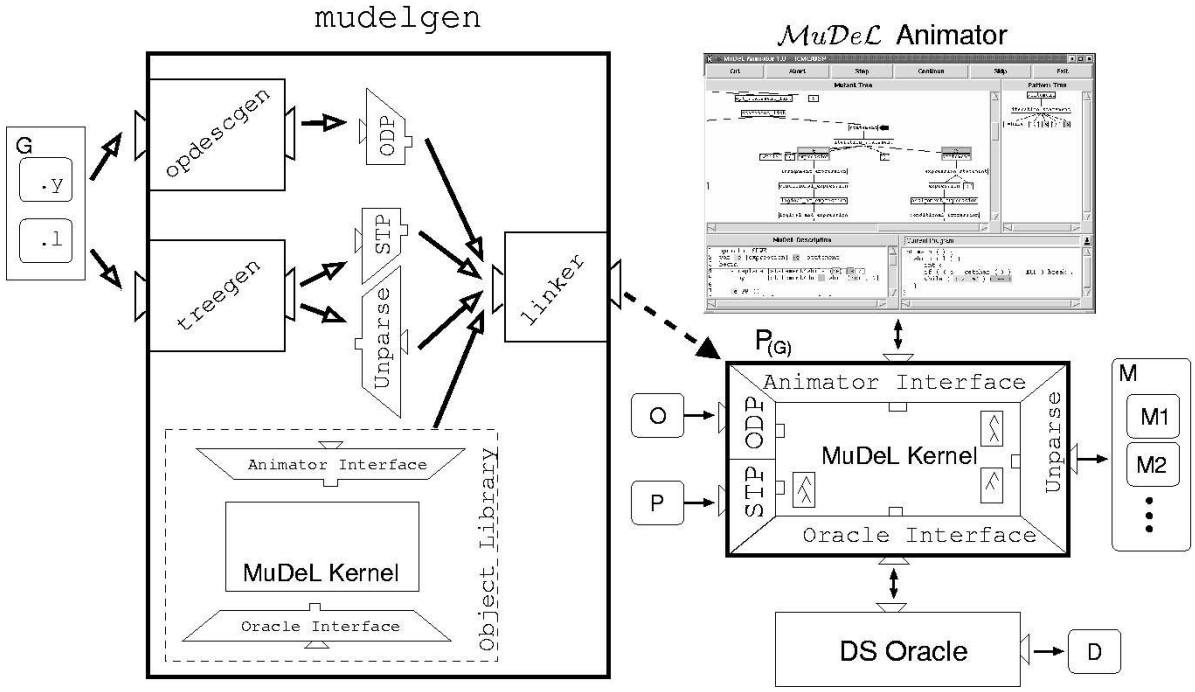

Figure 1: `mudelgen` Execution Schema

There are some units that will compose a $P_{(G)}$ that do not actually depend on G. We embodied these units in the `Object Library`. The greatest portion of the `Object Library` is devoted to the so-called MuDeL Kernel, which is responsible for interpreting the mutant operator description and manipulate the syntax tree accordingly. The remaining units in the `Object Library` allow the communication between the MuDeL Kernel and the external modules $\mathcal{MuDeL}$ Animator and DS Oracle, described in later sections.

The units that depend on the grammar are built by either `treegen` or `opdescgen`. Module `treegen` analyzes G and generates the units: (i) `STP` (Syntax Tree Processor), which is responsible to syntactically analyze a source product P and convert it 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 O 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` to generate the program $P_{(G)}$.

The program $P_{(G)}$ can then be run with a source product P and a mutant operator description O as input data. 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 mutated syntax trees, which are processed by `Unparser` 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) stored 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 verify whether the mutants were correctly generated (see Section 4). Moreover, the execution of the program $P_{(G)}$ can be visually inspected with the $\mathcal{MuDeL}$ Animator (see Section 5).

4 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, due mainly to the amount of output yielded. To cope with this problem, we adopted an approach that can be summarized in two steps. Firstly, we employed *denotational semantics* [2] to formally define the semantics of $\mathcal{MuDeL}$ 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 this formalism, to code and run the denotational semantics of $\mathcal{MuDeL}$. We implemented an external module DS Oracle that can be run in parallel with $P_{(G)}$ through the `Oracle Interface` in a *validation* mode. When invoked, the DS Oracle receives the information about a mutant operator O and derives a denotational function φ (in the mathematical sense) that formally defines the semantics of O. 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 φ. 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 where 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 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. While we employed an imperative stack-based approach in C++, we extensively used continuation and mappings [2] 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 $\mathcal{MuDeL}$ Animator

We have also implemented a prototyping graphical interface — called $\mathcal{MuDeL}$ Animator — for easing the visualization of a mutant operator execution. $\mathcal{MuDeL}$ Animator was implemented in Perl/Tk and currently has some limited features that allows to inspect the log of execution, without being able to interfere in the process. Figure 2 presents the main window of $\mathcal{MuDeL}$ Animator. At the top of the window are the buttons that control the execution of the animator, such as *Step*, *Exit* etc. The remaining of the window is divided up into four areas:

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

Mutant Tree: In the left top area, the animator shows the syntax tree of the product, reflecting any changes 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-variables 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 $\mathcal{MuDeL}$ **Description**) in the $\mathcal{MuDeL}$ Description area.

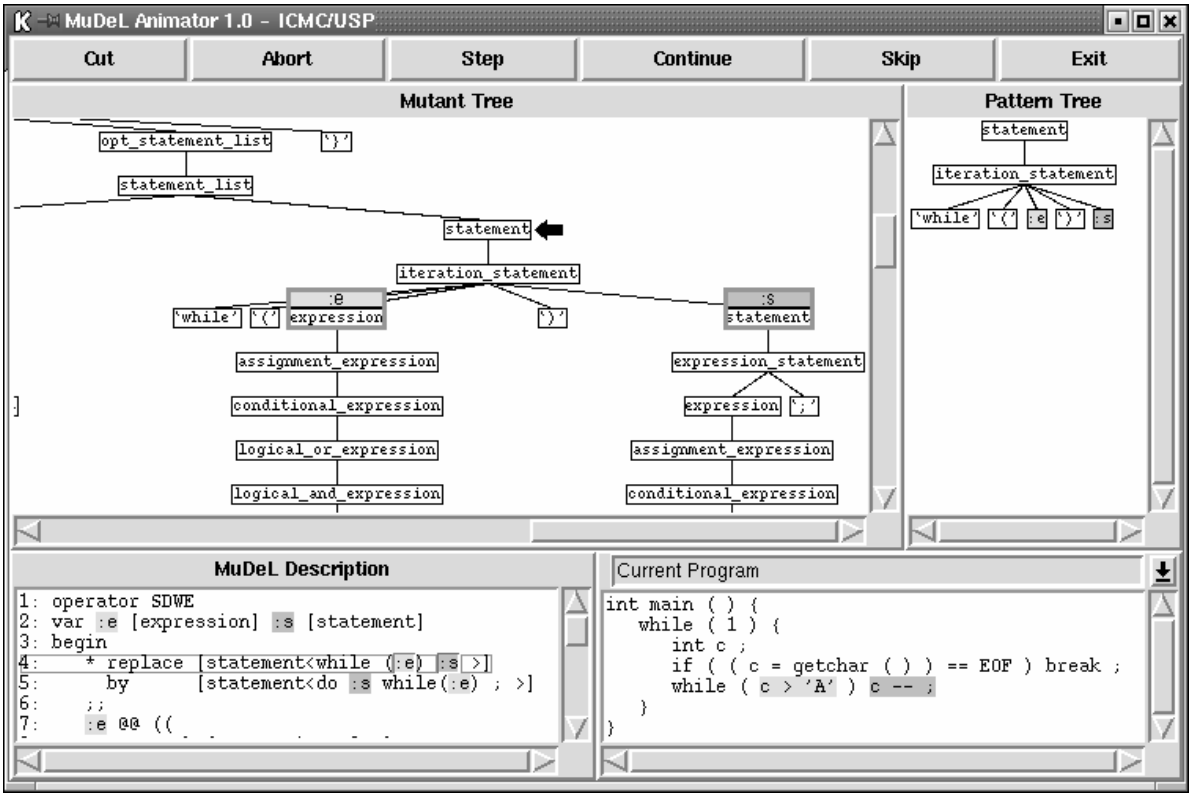

Figure 2: $\mathcal{MuDeL}$ Animator Main Window.

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.

6 Concluding Remarks

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

In this paper we briefly described `mudelgen`, a transformational-based system for generating mutants from $\mathcal{MuDeL}$ mutant operator definitions. $\mathcal{MuDeL}$ and `mudelgen` together form a powerful instrument in developing and validating mutant operators.

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

specifications and we are currently working on mutant operator descriptions for C++ and Java programs and coloured Petri net specifications.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

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

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

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

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

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

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

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

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

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

[11] A. S. Simão and J. C. Maldonado. 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, October 2001.

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