Mutation-based Testing of Model Transformations (Using HOT)

Ali Parsai

University of Antwerp
Antwerpen, Belgium
ali.parsai@student.uantwerpen.be

Abstract

Any software system is subject to failure. Testing is an important part of development cycle for any software, and it has a huge impact on the final product reliability. The classical point of view is inadequate to handle Model-Driven development process. That's why we need different methods to test such systems. We must determine if the test-suite is adequate for finding faults inside a system to be able to compare different test-suites. Mutation Testing is proven to be a convincing way to check the efficiency of a test-suite in a real-life situation. In Mutation Testing, a test-suite is run on different versions of a program (mutants) which include automatically generated deliberate faults. The result of this experiment shows the quality of the test-suite in detecting faults inside a program. To be effective, the generated faults should cover real-life situations. To generate these mutants, mutant operators are run on the original program. Previously, these operators acted on code-level which means that they must be implemented outside of the scope of the program itself. In this project, we are going to use Higher-Order Transformations to create mutant operators, so that the process of Mutation Analysis also correspond to Model-Driven development principles.

1. Introduction

Any software system is subject to failure. The parts of the code which causes failure is called a bug. Testing is thus an important part of development cycle for any software, and it has a huge impact on the final product reliability.

The classical point of view is inadequate to handle Model-Driven development process. Usually the type of the faults which is found in this method is different from other methods of programming. Most of the time, problems arise during model transformations, in which the product of transformation is different from what we expect. Since those problems are usually in the semantics level, to find and fix such problems we need to re-think the testing methods that is going to be used.

To advise on a certain method of testing for a specific case, first it must be determined if the test-suite is adequate for finding faults inside a system. There are several ways to find out the quality of a test-suite. Structural coverage is one such criteria which has been used before; yet, it is not directly related to the capacity of a test-suite to detect errors [2].

However, Mutation Testing is proven to be a convincing way to check the efficiency of a test-suite. In fact, it is the most reliable method to simulate real-life situations [1]. In Mutation Testing, a test-suite is run on different versions of a program (mutants) which include automatically generated deliberate faults. The result of this experiment shows the quality of the test-suite in detecting faults inside a program. To be effective, the generated faults should mimic the faults that can happen in the real-life situations [2].

To generate these mutants, mutant operators are run on the original program. Previously, these operators acted on code-level which means that they must be implemented outside of the scope of

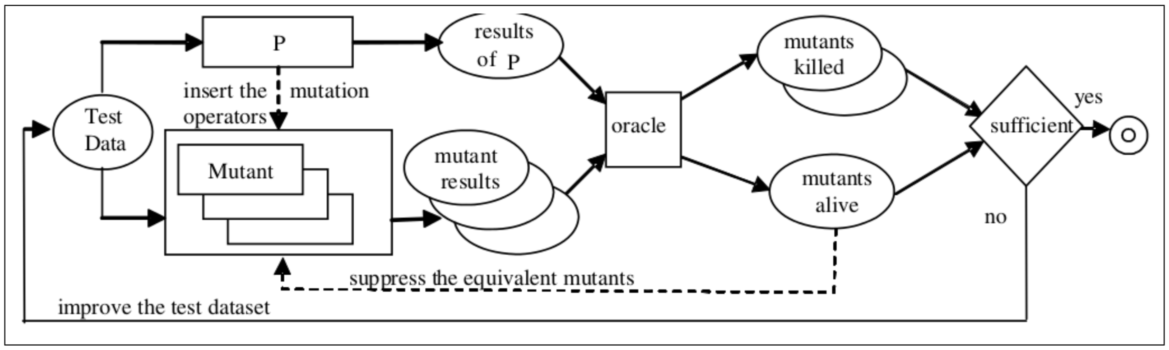

Figure 1: Mutation Process

the program itself.

In this project, we are going to use Higher-Order Transformations to create mutant operators, so that the process of Mutation Analysis also correspond to Model-Driven development principles.

2. Mutation Testing

Mutation Analysis is a method of testing which aims to evaluate a test-suite. In this method, the main program is changed by introducing a single fault inside its code. The resulting program is called a *mutant*. After creating a set of mutants, the test-suite is run on each mutant. If the test-suite fails to detect the mutant, it means that the failure could not be detected by the test-suite. Such mutant is called an *alive* mutant. If the test-suite finds the mutant, however, then the mutant is *killed*.

To create the mutants systematically, faults are modeled as mutant *operators*. When a mutant operator is applied to a program, a mutant is created. A test-suite is adequate if it can distinguish the program from all its non-equivalent mutants [2]. A mutant is equivalent to the program if they produce the same output for every possible input.

The result of the Mutation Analysis can be quantified by the percentage of killed mutants to non-equivalent mutants. The reliability of such result depends on the quality of the operators that are used to generate the mutants. These operators should be able to generate a wide range of possible faults to simulate real-life situations. Therefore, classical mutation operators are not relevant to Model-Driven development; since they only work on a syntactic level, and are not designed with Modeling-specific faults in mind.

3. Mutation Testing for Model-Driven Development

Classical mutation operators are not suitable for model transformation programs. Therefore we need to redefine some aspects of Mutation Analysis to be able to use it in this context. Figure 2 illustrates the model transformation process. The mutation operators should produce such mutants that can process inputs that conform to input meta-model and produce output that conforms to output meta-model.

Due to following reasons, classical operators are not relevant to this context [2]:

- *Mutant Significance*: a faulty model transformation may be different from a correct one in a complicated way, and not just by a single faulty statement.

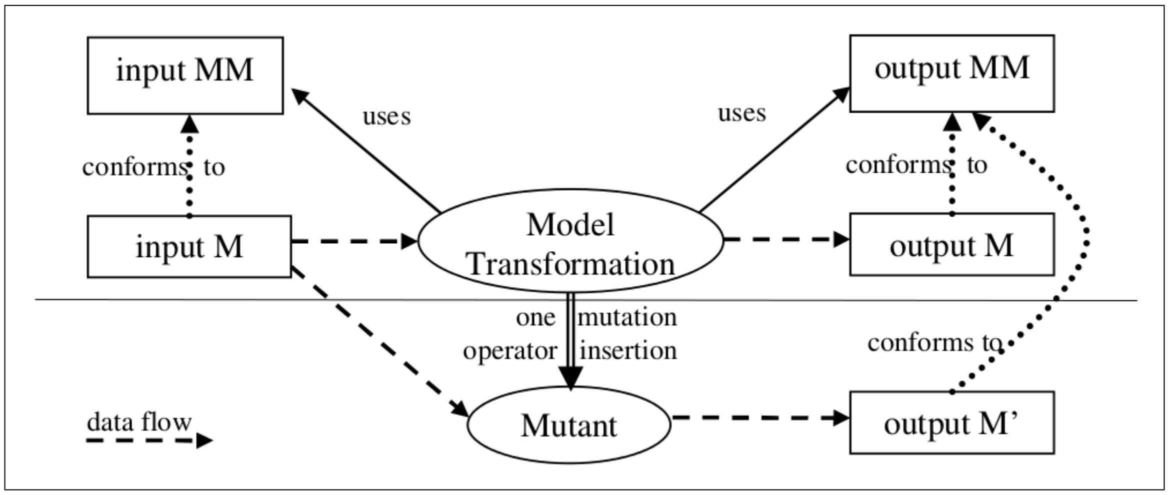

Figure 2: Model transformation process and a model transformation mutant creation

- *Mutant Viabilty*: a simple fault has a high probability to generate a non-viable mutant program.
- *Implementation Language Independency*: this operators work on syntactic part of the transformation instead of the semantic part, and therefore are dependent on the syntax of the implemented language.

So, we need operators which take model transformation into account. These operators are called semantic operators.

There are 4 categories of operators related to model transformations [2]. These categories correspond to the abstract operations generally performed during composition of a model transformation.

- **navigation:** the model is navigated and a set of elements is obtained.
- **filtering:** a treatment is applied on a subset of the previous set, so selection is done based on a filtering property.
- **output model creation:** output model elements are created from extracted elements.
- **input model modification:** when output model is a modification of the input model, input elements are changed.

Based on the above observation, several operators are defined which are dedicated to model transformations [2].

3.1 Navigation

- **Relation to the same class change (RSCC):** This operator replaces the navigation of one association towards a class with the navigation of another association to the same class (when the metamodel allows it).
- **Relation to another class change (ROCC):** This operator replaces the navigation of an association towards a class with the navigation of another association to another class.

- **Relation sequence modification with deletion (RSMD):** During the navigation, the transformation can navigate many relations successively. This operator removes the last step off from the composed navigation.
- **Relation sequence modification with addition (RSMA):** This operator does the opposite of RSMD. The number of mutants created depends on the number of outgoing relations of the class obtained with the original transformation.

3.2 Filtering

- **Collection filtering change with perturbation (CFCP):** This operator aims at modifying an existing filtering, by influencing its parameters. One criterion could be a property of a class or the type of a class; this operator will disturb this criterion.
- **Collection filtering change with deletion (CFCD):** This operator deletes a filter on a collection; the mutant returns the collection it was supposed to filter.
- **Collection filtering change with addition (CFCA):** This operator does the opposite of CFCD. It uses a collection and processes a useless filtering on it. This operator could return an infinite number of mutants, we have to restrict it. We choose to take a collection and to return a single element arbitrarily chosen.

3.3 Creation

- **Class compatible creation replacement (CCCR):** This operator replaces the creation of an object by the creation of an object of a compatible type. It could be an instance of a child class, of a parent class or of a class with a common parent.
- **Classes association creation deletion (CACD):** This operator deletes the creation of an association between two instances.
- **Classes association creation addition (CACA):** his operator adds a useless creation of a relation between two class instances of the output model, when the metamodel allows it.

4. Generating Mutant Operators Using Higher-Order Transformations

A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models [4]. The popularity Model-Driven development as a main method of creating new frameworks makes conformity of any testing method very important. This fact leads to avoiding the use of code-based mutation operator definitions. The availability of concepts like Higher-Order Transformations means that we can implement the already available ideas in this field in a new way that is more suitable to the whole concept of Model-Driven Development. Therefore, the aim of this project is to implement the mutation operators using HOTs.

To achieve this goal, the mutation operators are applied to the transformation rules as a transformation themselves. Currently, the only tool which can produce such models is AToMPM ([3]). After generating the mutants, we use the same process in Figure 2 to generate the faulty output models.

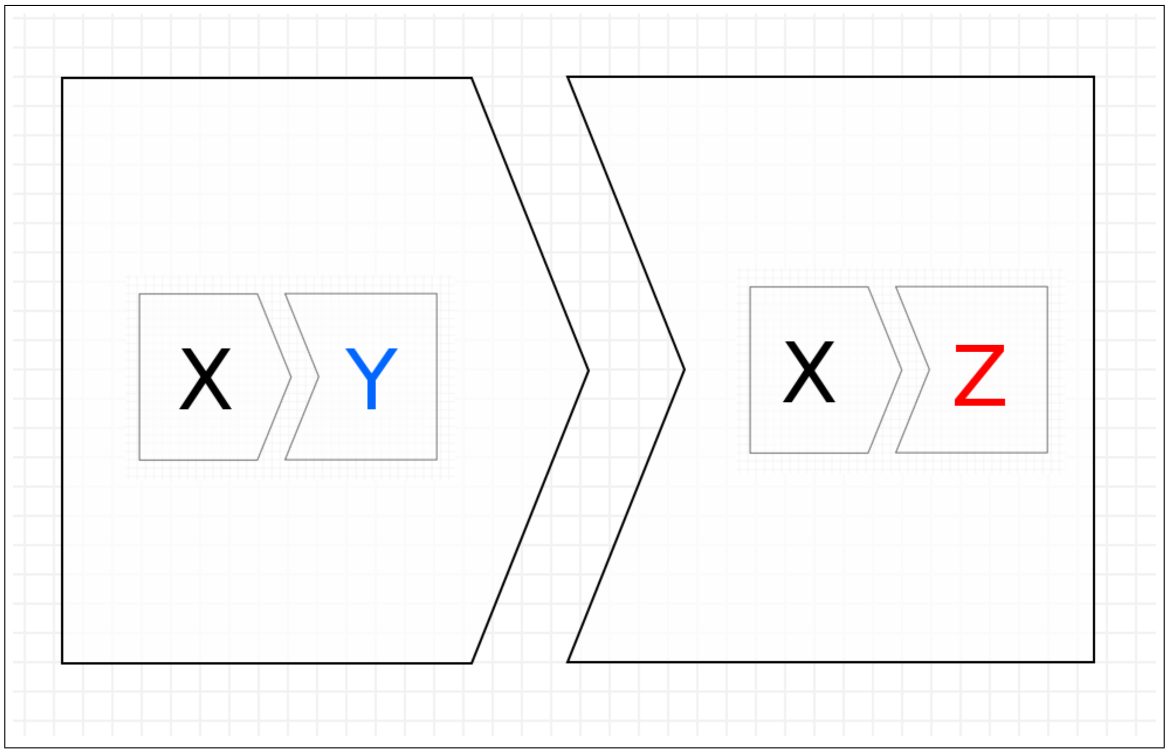

Figure 3: Mutation Operator modeled as Higher-Order Transformation

References

[1] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411, 2005.

[2] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *Model Driven Architecture–Foundations and Applications*, pages 376–390. Springer Berlin Heidelberg, 2006.

[3] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. MODELS.

[4] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Model Driven Architecture-Foundations and Applications*, pages 18–33. Springer, 2009.