# MDE Reading report:
# Mutation-based testing of model transformations

Joran Dox[a]

[a]*University of Antwerp*

**Abstract**

To ensure the quality of software, it needs to be tested. Of course, the used test suite requires testing in itself. In mutation-based testing, this testing suite is tested with injecting common mistakes into the software, and checking whether those are found in the tests. In the context of Model Driven Engineering, testing model transformations is done by mutating the transformation model through the use of Higher-Order Transformations.

*Keywords:* testing, mutation, mutation-based testing, Model Driven Engineering, Higher-Order transformations, model transformations, metatesting

## 1. Introduction

*"Quis custodiet ipsos custodes?" - Juvenal, Satire VI*

In software engineering, testing the software is an integral part of the development process. It then follows that this testing suite should itself be tested too. One way of testing this is with the use of mutation-based testing. This technique entails changing the software in ways that resemble common errors, and checking whether the testing suite can differentiate between the mutations and the original software.

In Model Driven Engineering (MDE), model transformations are used to transform models, much like in classic software engineering the software transforms the data. Applying mutation-based testing to model transformations would then mean transforming the original transformation with common errors, before running the test suite. The transformations applied to these transformations are called Higher-Order Transformations (HOT).

In this paper, these concepts are explained a little more in-depth, based on reading from the papers cited.

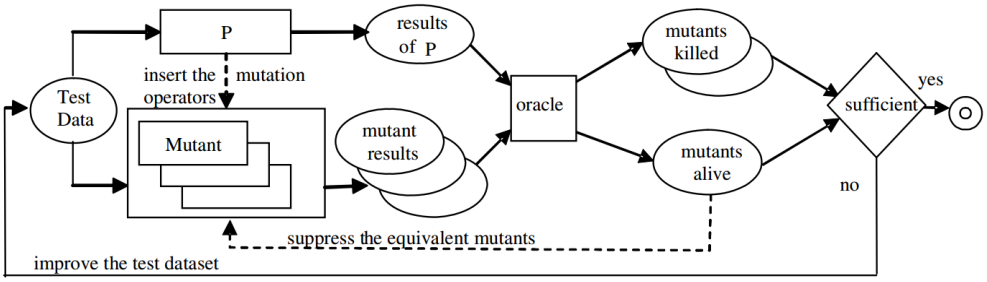
## 2. Mutation-Based Testing

The concept of mutation-based testing was first explored in classic software engineering as a way to test a test suite [1]. This approach takes the source code of a program and changes small things. To be effective, the changes made should reflect common mistakes:

*Email address:* joran.dox@student.uantwerpen.be (Joran Dox)

- add / subtract 1 from integer constants
- change * to /
- change TRUE to FALSE
- delete a statement
- ...

The resulting source code, assuming it compiles, is then run through the test suite. If the test suite does not find the program faulty, then either a new test should be added to address this (faulty test suite) or the mutation was ineffective for some reason (e.g. dead code or redundant checks).




In this graph, courtesy of [2], P is the program to be tested, the testing suite is in the form of the test data (input) and the oracle, which "kills" the detected mutants. Any mutants left over that are not equivalent to P are then used to improve the test dataset.

## 3. Model Transformations and Higher-Order Transformations

In Model-Driven development, we can draw a parallel between the input/output of a program and a model, and between software and model transformations. It then follows that software taking software as input and producing mutated software as output, has its equivalent in model transformations applying to model transformations. This kind of meta-transformations are called Higher-Order Transformations [3].

## 4. Mutation-based Testing of Model Transformations

When applying the concept of mutation-based testing to MDE, one problem arises immediately: what mutations should be created to reflect real-life situations? Common mistakes in classical programming are not necessarily common mistakes in MDE. [2] defines following categories:

- *navigation:* the model is navigated thanks to the relations defined on its input/output meta-models, and a set of elements is obtained.
- *filtering:* after a navigation, a set of elements is available, but a treatment may be applied only on a subset of this set. The selection of this subset is done according to a filtering property.

- *output model creation:* output model elements are created from extracted element(s).
- *input model creation:* when the output model is a modification of the input model, elements are created, deleted or modified.

Across these categories, 10 mutation operators are defined for a model transformation context. The list is moved to Appendix A, copied from [4] for brevity, but can be found in full detail in paper [2].

## 5. Conclusion

Mutation-based testing is a great way to ensure the reliability of a test suite in classic software engineering. It also holds potential to be applied in Model-based development. Implementing a mutation-based meta-testing suite in AToMPM [5] would be an asset to the MDE community, and this will be my goal in the following project.

## Appendix A. Mutant Operators

### *Appendix A.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.

### *Appendix A.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 toreturn a single element arbitrarily chosen.

*Appendix A.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.

## References


[1] J. Andrews, L. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? [software testing], in: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, 2005, pp. 402–411. doi:10.1109/ICSE.2005.1553583.

[2] Mutation analysis testing for model transformations, in: A. Rensink, J. Warmer (Eds.), Model Driven Architecture Foundations and Applications, Vol. 4066 of Lecture Notes in Computer Science, 2006. doi:10.1007/11787044_28.

[3] On the use of higher-order model transformations, in: R. Paige, A. Hartman, A. Rensink (Eds.), Model Driven Architecture - Foundations and Applications, Vol. 5562 of Lecture Notes in Computer Science, 2009. doi:10.1007/978-3-642-02674-4_3.

[4] A. Parsai, Mutation-based testing of model transformations (using hot), [Online; accessed 10-December-2015, `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201314/projects/Ali.Parsai/`] (2013).

[5] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, H. Ergin, Atompm: A web-based modeling environment., in: Demos/Posters/StudentResearch@ MoDELS, 2013, pp. 21–25.