MDE Project 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. The proposed mutation operators are also explored with AToMPM[1].

2. Reading

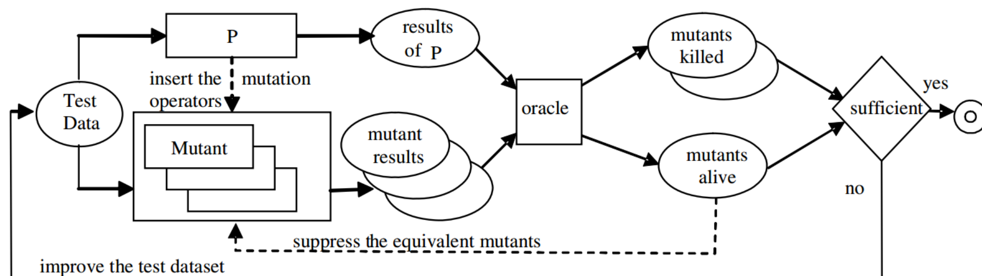
2.1. Mutation-Based Testing

The concept of mutation-based testing was first explored in classic software engineering as a way to test a test suite [2]. 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 [3], 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.

2.2. *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 [4].

2.3. *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. [3] 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 can be found in full detail in paper [3], and is explained further in this paper in section 3.3.

2.4. 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 [1] would be an asset to the MDE community, and exploring this will be my goal in the following project.

3. Project

3.1. AToMPM

AToMPM stands for "A Tool for Multi-Paradigm Modeling". It is a research framework from which you can generate domain-specific modeling web-based tools that run on the cloud. AToMPM is an open-source framework for designing DSML environements, performing model transformations, and manipulating and managing models. It runs completely over the web, making it independent from any operating system, platform, or device it may execute on. AToMPM follows the philosophy of modeling everything explicitly, at the right level of abstraction(s), using the most appropriate formalism(s) and process(es), being completely modeled by itself.[5]

For this project, a modified version of AToMPM is used. The most important difference is that Higher Order Transformations are supported. To make them possible, an explicit RAMification transformation is created.

As a quick refresher: RAMification stands for Relax, Augment and Modify; three actions which need to be applied to a metamodel to create a pattern metamodel which can be used to create model transformations. Relaxing constraints like minimum cardinalities, augmenting the metamodel with labels and a concrete syntax for abstract classes, and modifying attributes to enable filtering based on the model's attributes.

This RAMification transformation is used as follows: to ramify a formalism, both the metamodel and concrete syntax are loaded alongside each other, after which the transformation is run. The result is saved under *formalism*.ramified.model and then compiled with the regular CompileMenu toolbar into *formalism*.ramified.metamodel and *formalism*.ramified.defaultIcons.metamodel. These can be used exactly like the *.pattern* metamodels used in the current version of AToMPM, as they are also one level of RAMification. To apply another level of ramification, treat the *formalism*.ramified.model as the initial metamodel and apply the RAMification transformation again, saving the result as *formalism*.ramified.ramified.model. This can then again be compiled to a metamodel and concrete syntax.

3.2. Preparation and initial troubles

As is often the case with research software (and an experimental version of it no less), the initial stage of development of this project was hindered a lot through bugs, some of which were very hard to track down, let alone fix. A particularly nasty one was apparently related to localisation in Windows.

Another problem is the fact mutation-based testing is intended to improve a test suite, which currently does not exist for AToMPM. As such, while the transformations themselves may work, no experimental data can be gathered about the fitness for that purpose.

Inside of the modified AToMPM client, more had to be done to be able to use the HOTs. First of all, the "TransformationRule" formalism had to be RAMified itself, using the newly provided transformation. Next, any rule you wish to transform through HOTs needs to be modelled in the created *.ramified* metamodels, instead of using the existing *.pattern* versions. This is because the metamodels need to be ramified once per level of Higher Order, (in this case two), and *.ramified* and *.pattern* metamodels don't work together. Luckily the conversion from the existing *.pattern*-based transformation rules to *.ramified* versions is as simple as replacing the mentioned metamodels in the files, as both are functionally the same.

At this point we can start creating Higher Order Transformations using TransformationRule, TransformationRule.ramified and *formalism*.ramified.ramified.

3.3. Mutant Operators

The following Mutator Operators will be explained using the TrainSystem formalism used in the practice sessions of the MDE course[6] when appropriate. For brevity, the abbreviations Left Hand Side (LHS) and Right Hand Side (RHS) will be used for the *before* and *after* sides of a transformation respectively.

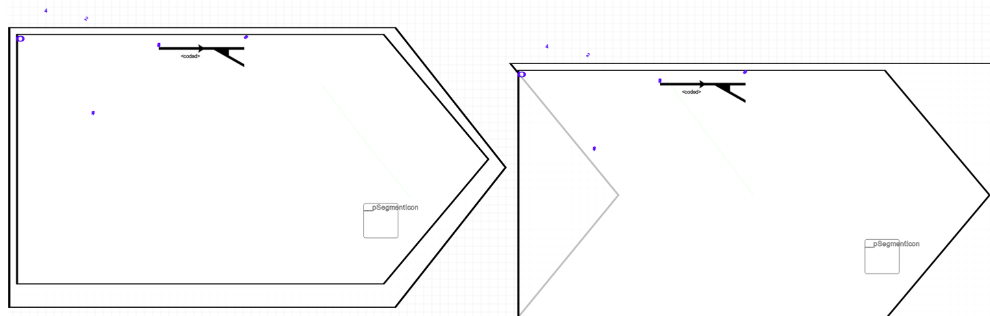
3.3.1. Navigation

In this section, the navigation part of the transformation is mutated. In AToMPM, this relates to changing the objects in the LHS, or the relations between them.

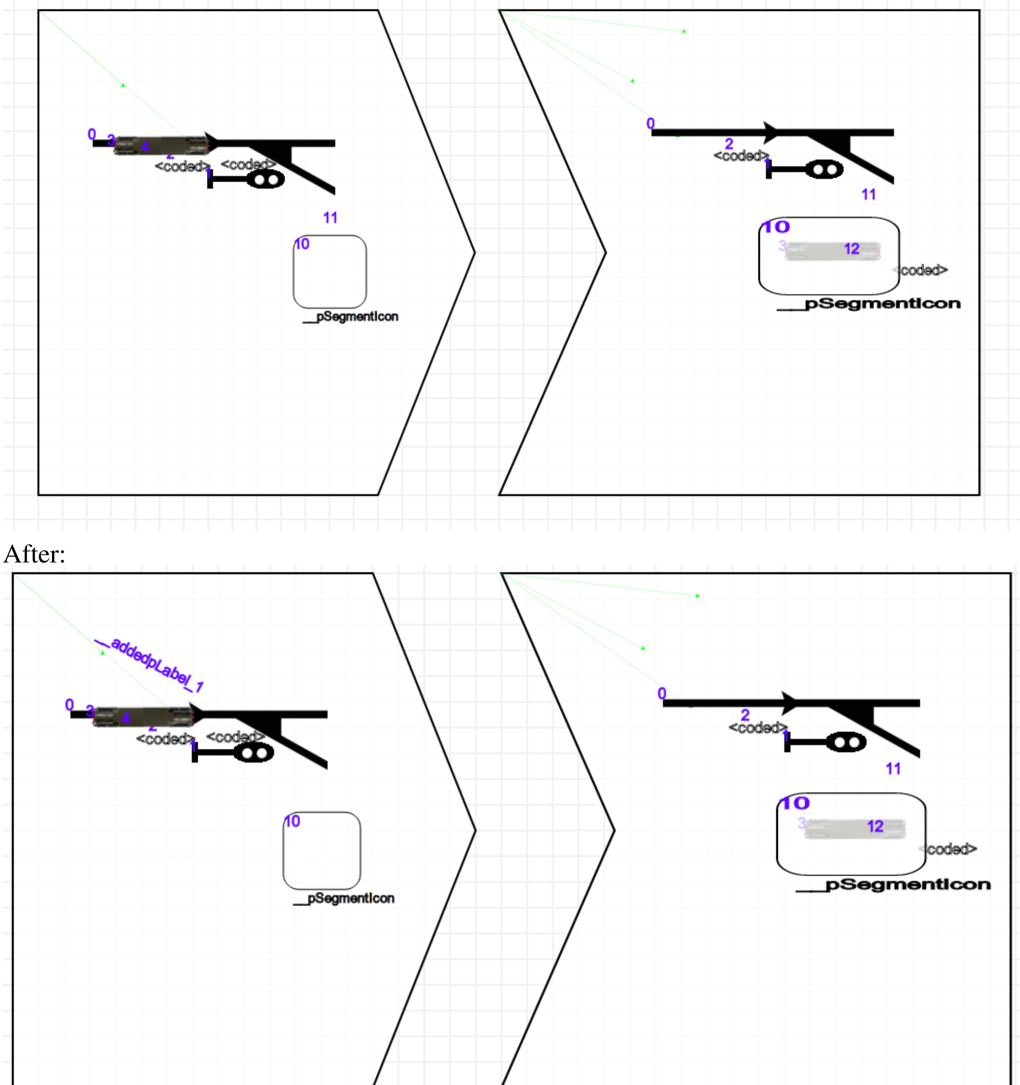
- *Relation to the same class change (RSCC): This operator replaces the navigation of one association towards a class with the navigation of another association to the same class (when the metamodel allows it).*[7]

  In this transformationrule, the train moves to the segment connected to the diverging link of the turnout. After applying the mutation, this diverging link is replaced with a straight link, resulting in a train moving straight although the turnout's direction is set to diverging. For a test suite to detect this, a generic test on a big network comparing expected end stations with real end stations would be sufficient.

  Mutator:



  Before:

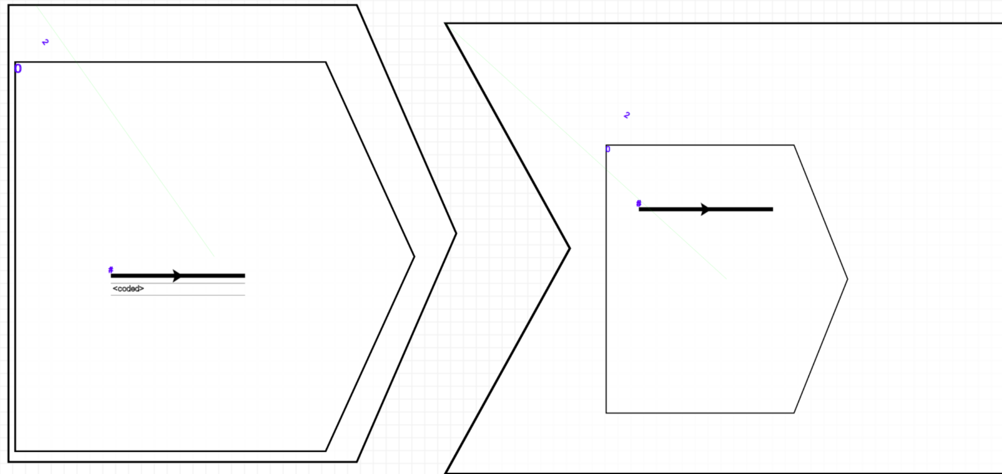


After:

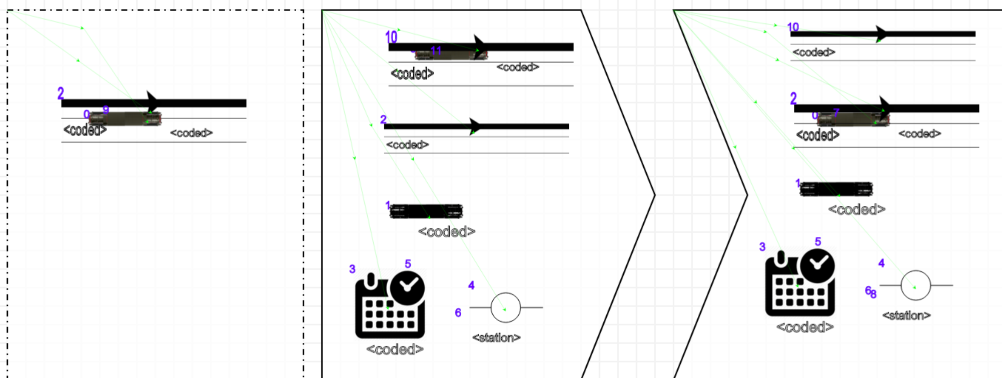
- *Relation to another class change (ROCC): This operator replaces the navigation of an association towards a class with the navigation of another association to another class.*[7]

In this transformationrule, a train is placed on the correct starting station, severing the connection with a previous station it might be on. The mutator replaces this previous station with a regular straight however. Any test starting with a train on a wrong station will show that this rule doesn't work correctly anymore.

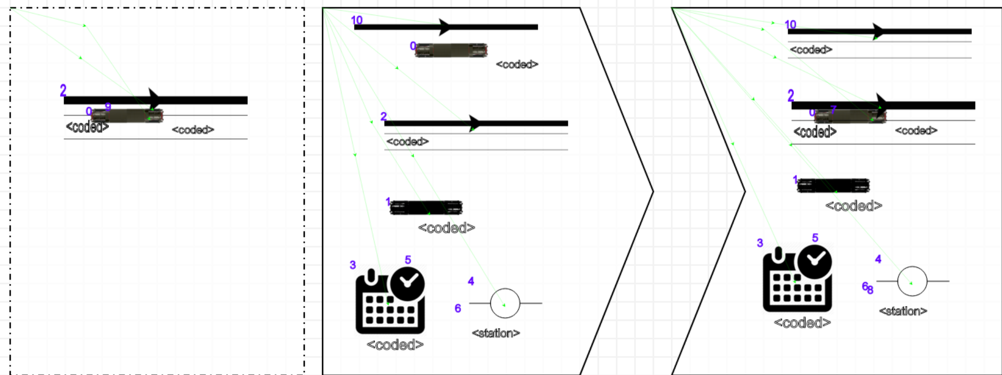
Mutator:



Before:



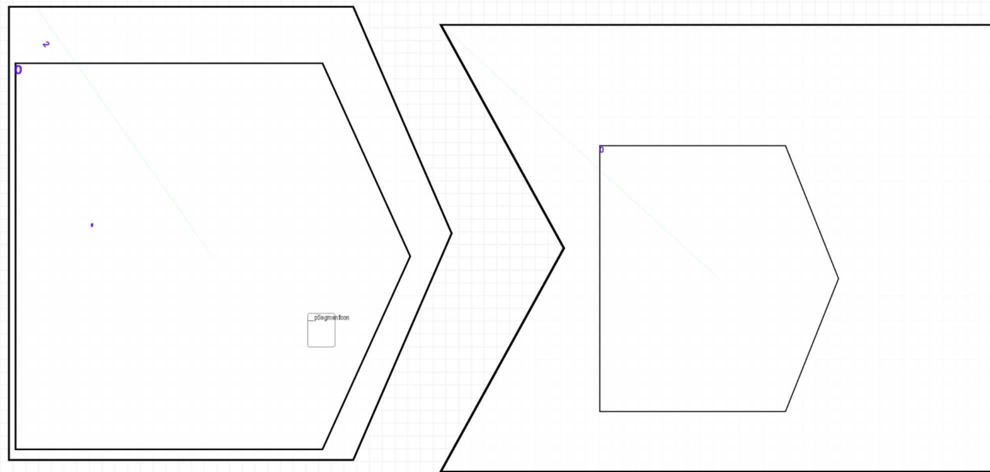
After:



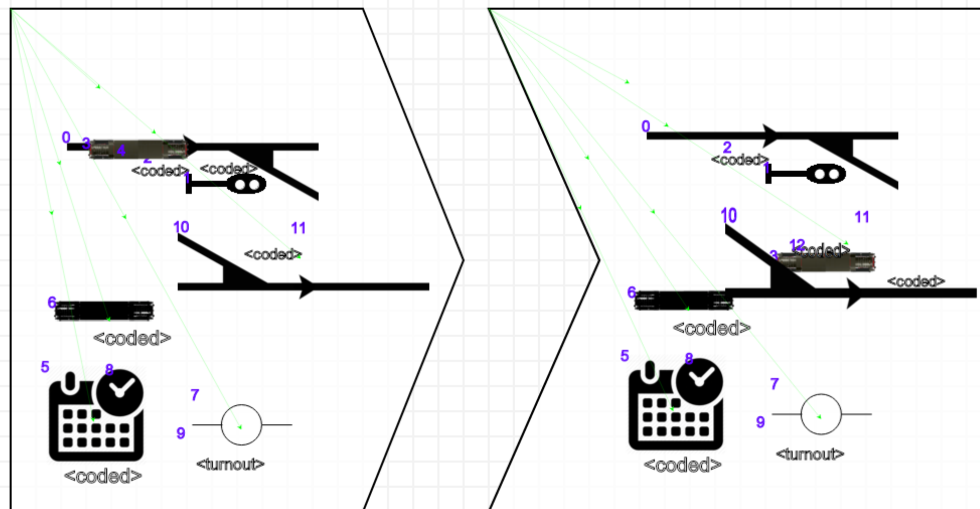
- *Relation sequence modification with deletion (RSMD): During the navigation, the transformation can navigate many relations successively. This operator removes the last step off from the composed navigation.*[7]

This rule moves a train from his turnout, set to diverging, to the junction it is connected to. Changing this rule moves a train from anywhere to a junction, invalidating many checks along the way. This resulting rule would probably not qualify as ”compilable” and as such the mutant is killed before any testing needs to be done. This is partly due to the absence of pivot elements, which would cause a new turnout to be created on the RHS if it is removed on the LHS.

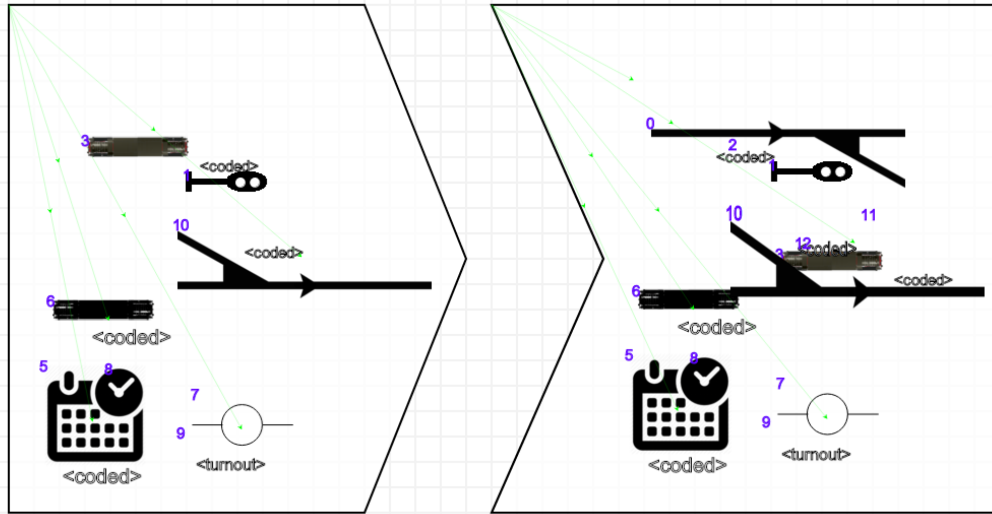
Mutator:



Before:



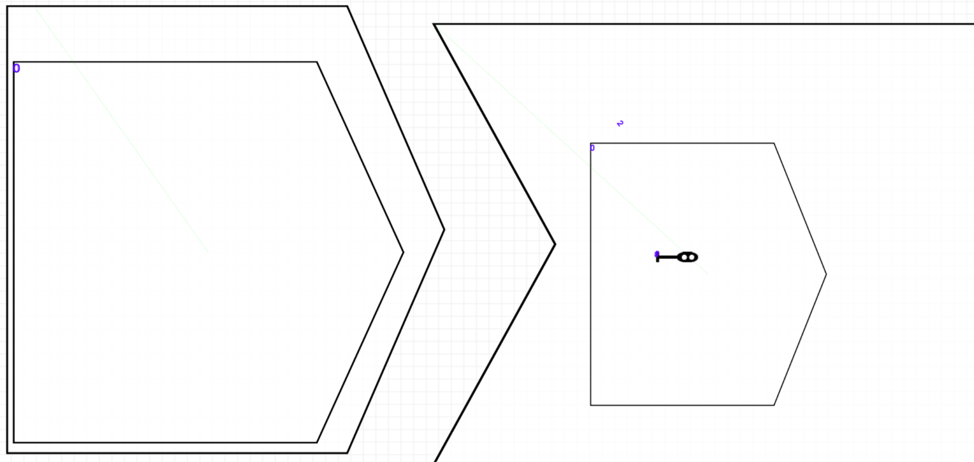
After:



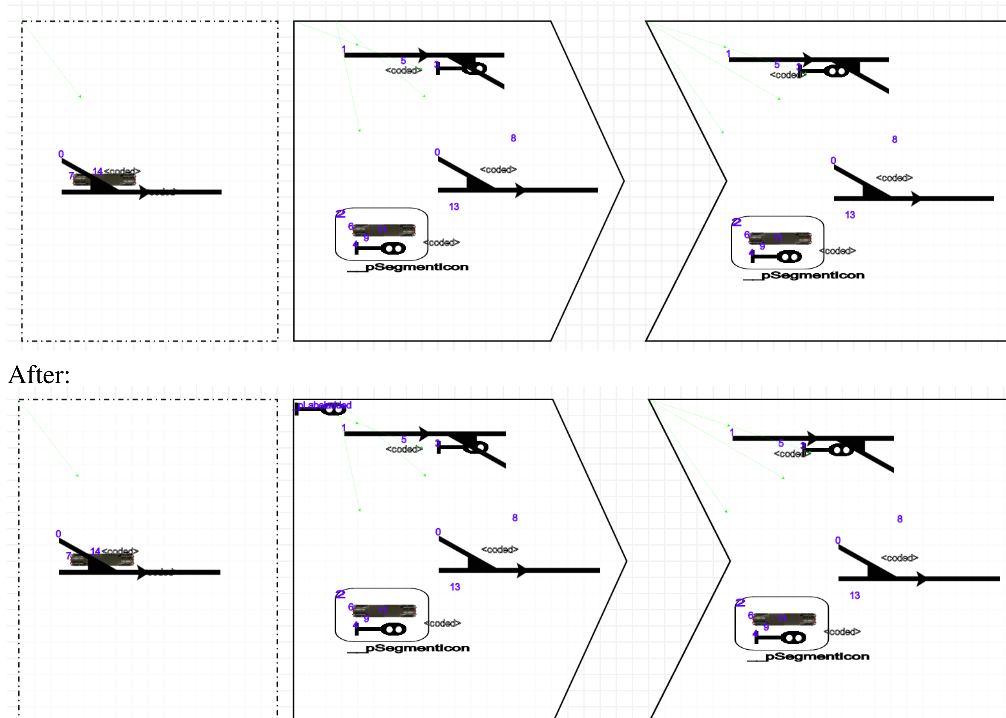
- *Relation sequence modification with addition (RSMA): This operator does the opposite of RSMD. The number of mutants created depends on the number of outgoing relations of the class obtained with the original transformation.*[7]

  In this rule, the light of a segment the train is on is set to green, after some safety checks. The mutation adds a random green light to be matched somewhere, which should not matter in any network, since the already-matched junction should have a light on it anyway. As such, this is an example of a mutation which is equivalent to the original.

  Mutator:



Before:



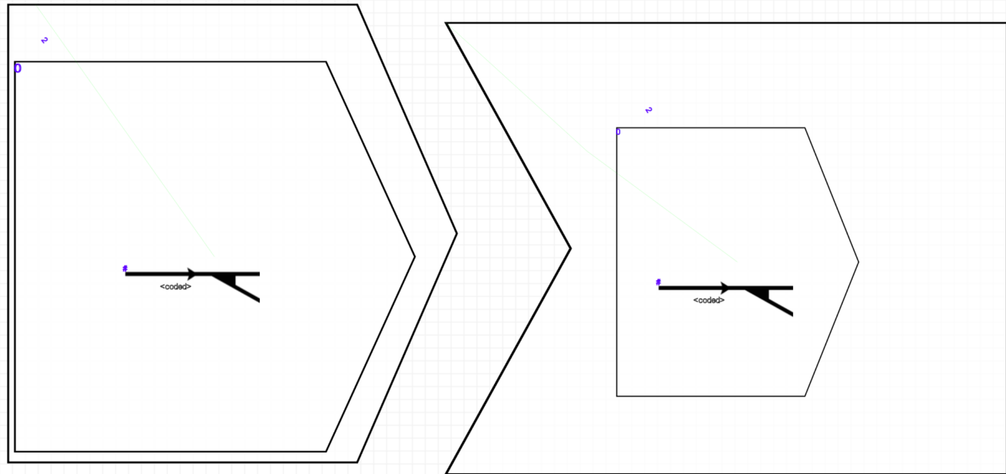
3.3.2. Filtering

In this section, the filtering part of the transformation is mutated. In AToMPM, this relates to changing the filters on the attributes of the objects and relations in the LHS.

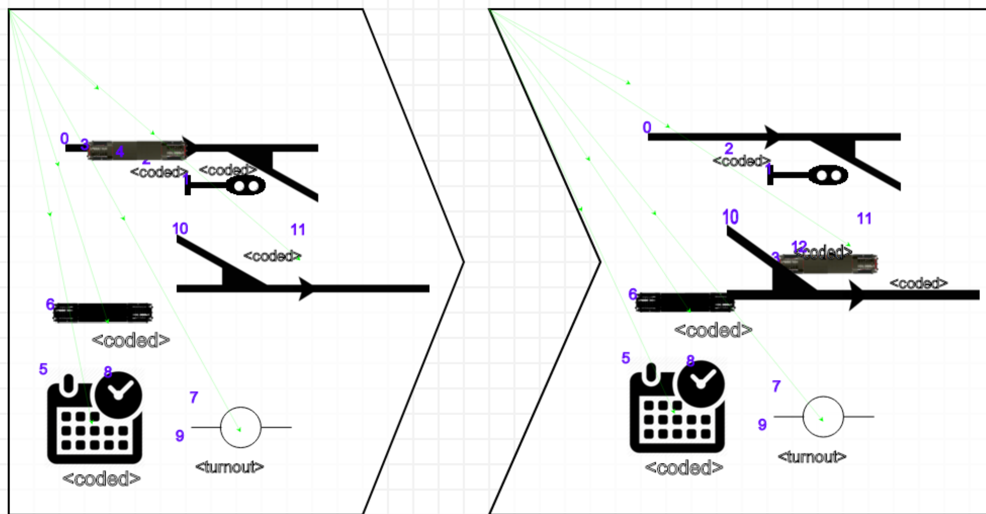
- *Collection filtering change with perturbation (CFCP): This operator aims at modifying an existing filtering, by influencing its parameters. One criterion could be a property of a class or the type of a class; this operator will disturb this criterion.*[7]

  This rule, the same one as in the RSMD section, moves a train from his turnout, diverging, to a junction. The mutated version however, changes this diverging property and turns it straight, causing trains to do the opposite of the turnout's setting and diverge when the setting is straight. This mutation is again simple to detect with a test checking the correct end stations of the trains, assuming the test network lets the situation presents itself.

  Mutator:



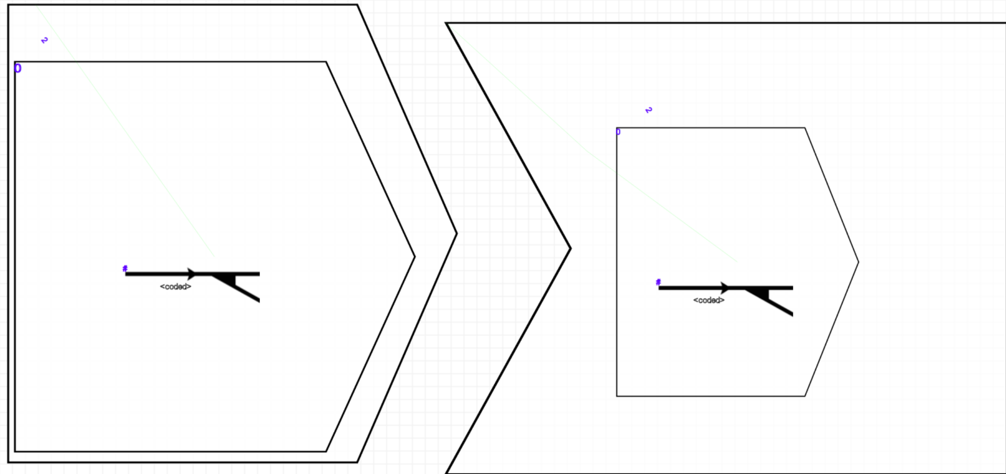
Before and after look the same:



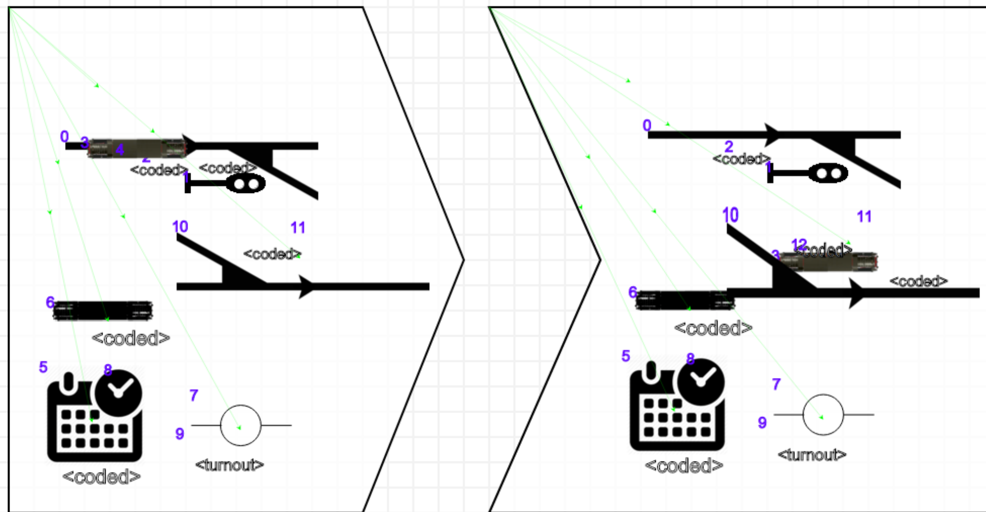
- *Collection filtering change with deletion (CFCD): This operator deletes a filter on a collection; the mutant returns the collection it was supposed to filter.*[7]

Again in the same rule, we can remove the filtering on direction entirely, resulting in trains always diverging on turnouts followed by junctions. Likewise, this mutant can be detected with an appropriate simulation.

Mutator:



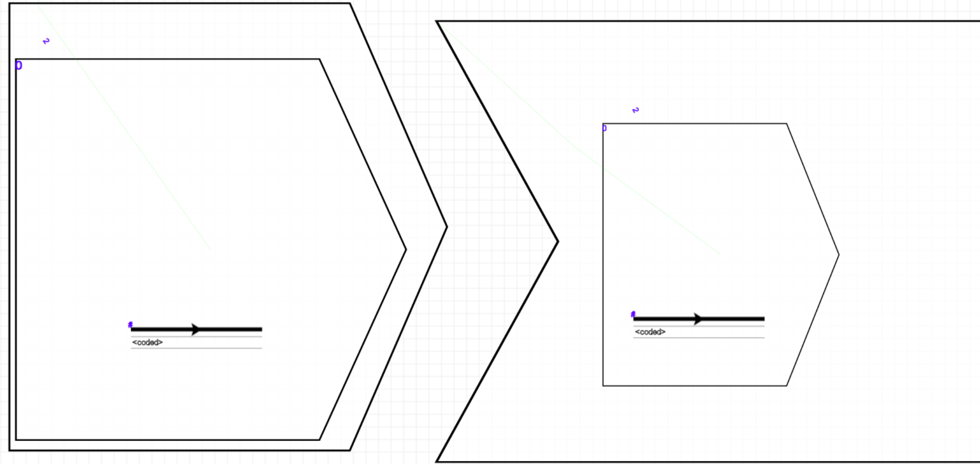
Before and after look the same:



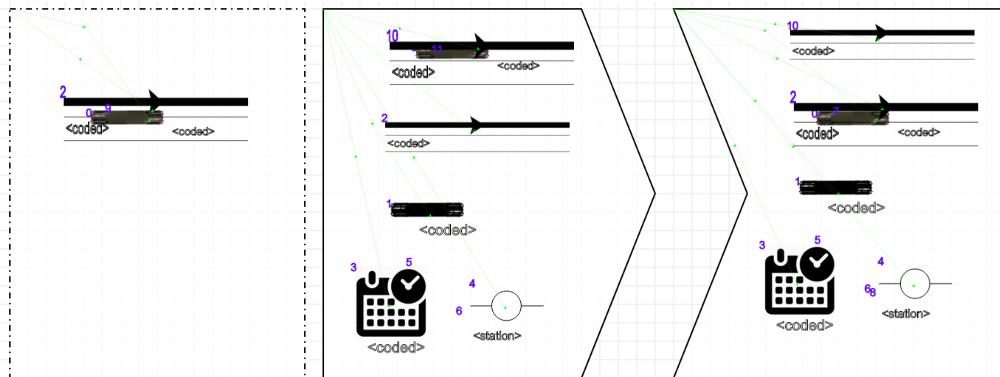
- *Collection filtering change with addition (CFCA): This operator does the opposite of CFCD. It uses a collection and processes a useless filtering on it.*[7]

In this rule, the same one as in the ROCC section, the train is moved from his original (wrong) station to the correct starting station according to its schedule. The mutator changes the name filter on one of the stations in the LHS to require its name to be ”Antwerp”. Evidently in most cases this rule will not find a match, and as such fail to run. Any test with random station names should detect that something is wrong and kill the mutant.

Mutator:



Before and after look the same:



3.3.3. Creation

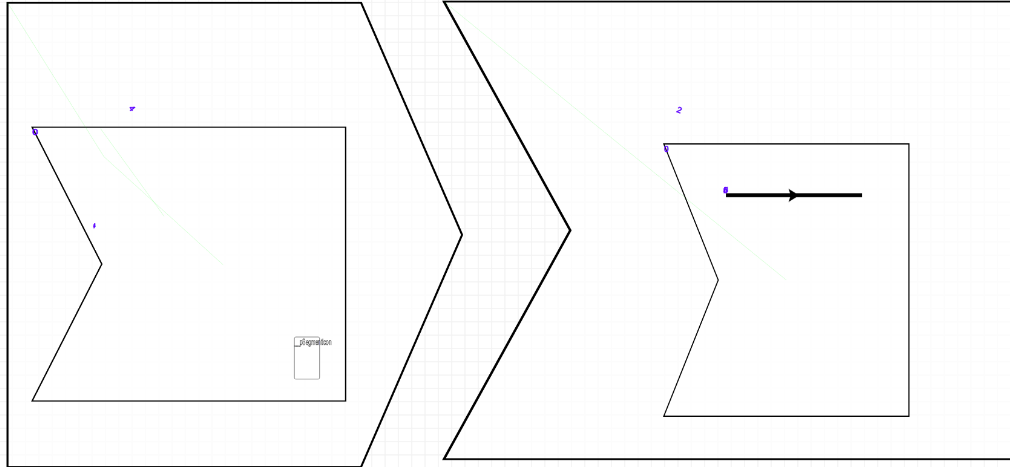
In this section, the creation part of the transformation is mutated. In AToMPM, this relates to changing the objects and relations in the RHS of the transformation.

Currently this part of the Higher Order Transformations does not work properly, and the following mutations are hypothetical:

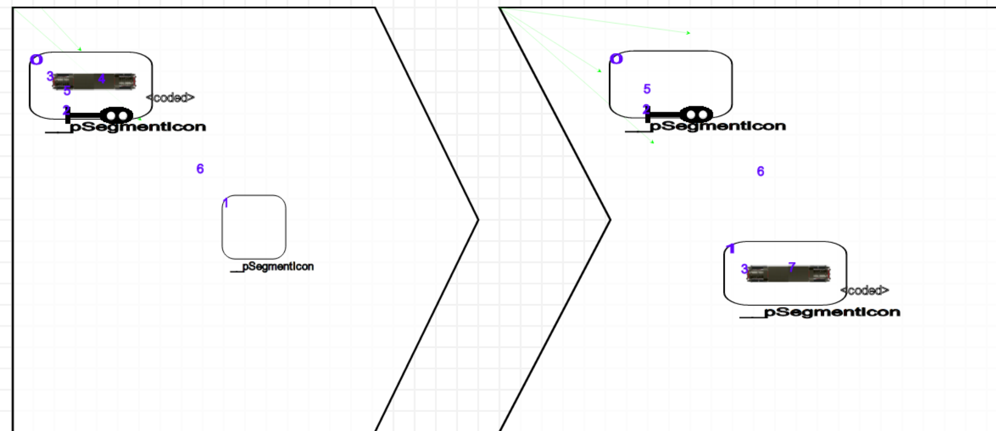
- *Class compatible creation replacement (CCCR): This operator replaces the creation of an object by the creation of an object of a compatible type. It could be an instance of a child class, of a parent class or of a class with a common parent.*[7]

  Strictly speaking the objects in the RHS are only created if they were not present in the LHS, but as this situation does not happen in the Trainsystem formalism, the broader interpretation of any object present in the RHS is used here. In this rule, the train moved from his current segment to the next one, given that the light is green. The mutator replaces a segment by a straight. As a result, the train transforms all segments it passes into straights, violating various constraints along the way. As such, this mutation would once again be found by any test that triggers the change.

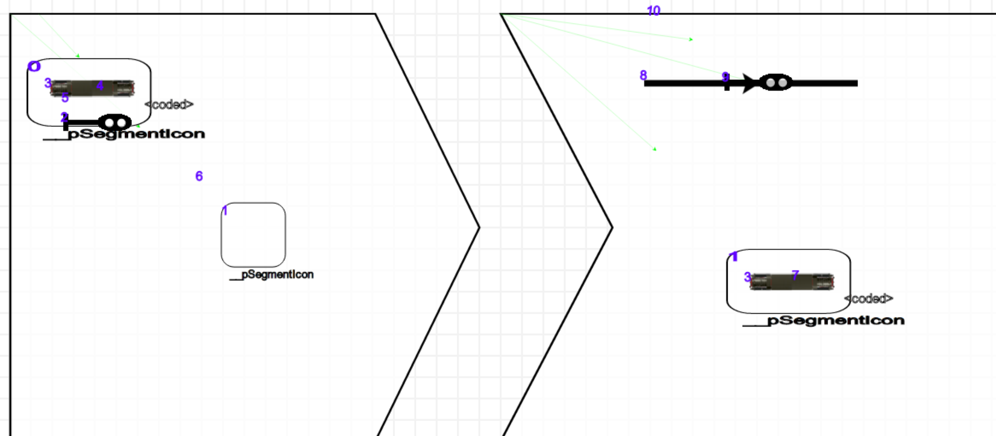
Mutator:



Before:



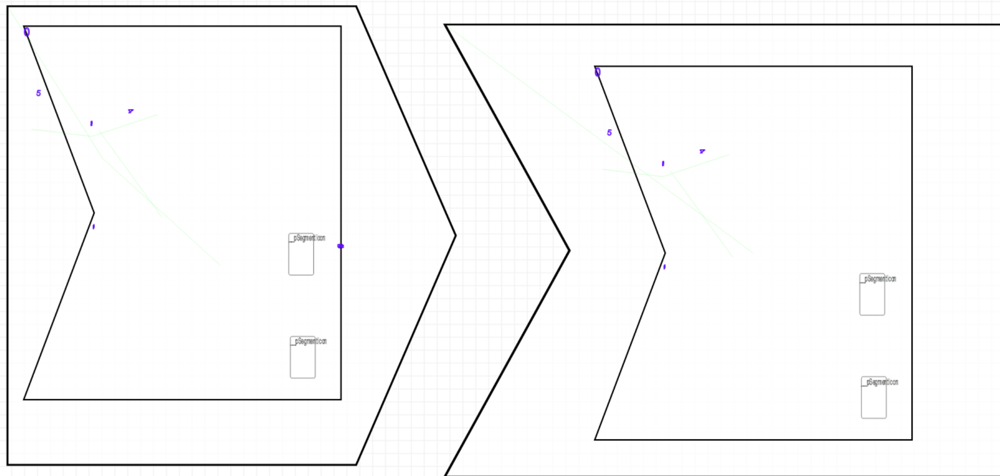
After:



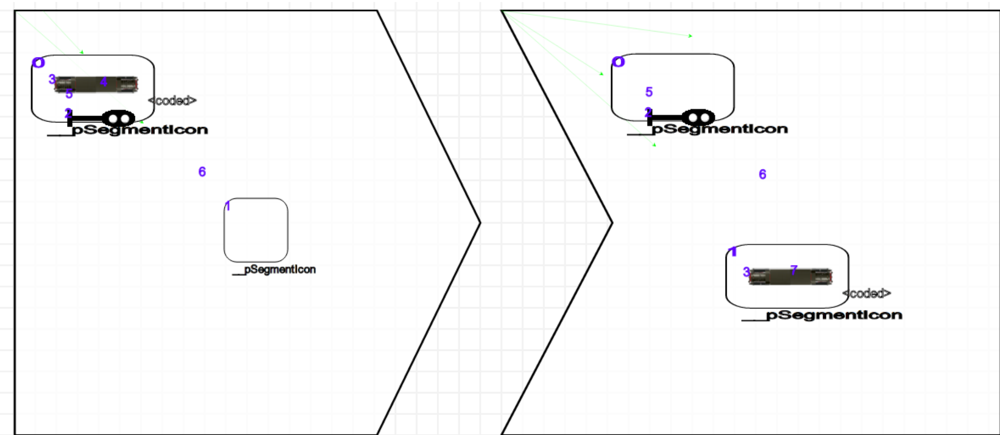
- *Classes association creation deletion (CACD): This operator deletes the creation of an association between two instances.*[7]

In the same rule, this mutator removes the link between the two segments, causing subsequent trains arriving there to stop seemingly without a reason. This one would violate minimum cardinalities unless it happens to sever a link between two stations.

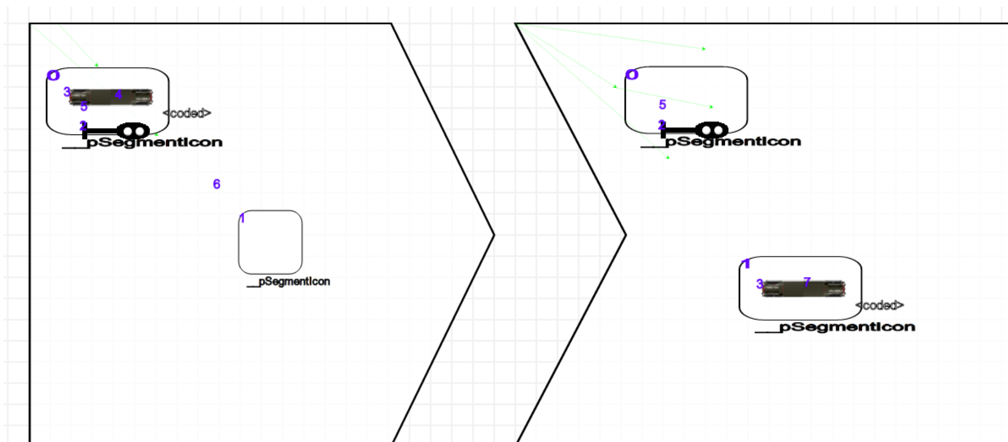
Mutator:



Before:



After:



- *Classes association creation addition (CACA): This operator adds a useless creation of a relation between two class instances of the output model, when the metamodel allows it.*[7]

  The used railway metamodel does not allow extra relations to be created in any way apart from connecting two completely separate railroad networks end-to-end. In any other case the cardinalities and/or other constraints would be violated.

4. Conclusion

As has been shown so often, testing is important, important enough to warrant being tested itself. In a model-driven environment, this is still true. In this project I explored the application of Higher Order Transformations in AToMPM for this goal. While I did not succeed to create a complete metatesting suite, I believe I have shown the potential this approach has with adequate examples.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] E. Syriani, H. Vangheluwe, AToMPM, [Online; accessed 21-January-2016, `http://www-ens.iro.umontreal.ca/ syriani/atompm/atompm.htm`] (2016).

[6] H. Vangheluwe, Model Driven Engineering, [Online; accessed 21-January-2016, `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/`] (2016).

[7] 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).