Mutation-based testing of rule-based model transformations (using HOT)

Ali Parsai

University of Antwerp

Abstract

Testing is an important part of the development cycle of a software. Mutation testing can be used to assess the quality of a test-suite in a quantifiable and repeatable way. Applying mutation testing to model-driven context requires rethinking of the classical approach. One way of implementing this idea in model-driven context is by using higher-order transformations as mutation operators for model transformations. In this article we explore this idea by implementing it on the RPG Game meta-model using AToMPM.

Keywords: mutation testing, model-driven, higher order transformations

1. Introduction

Since the dawn of software engineering, one of the problems developers faced was to be able to make sure their software is free of bugs. A bug is defined as a fault in the system which results in unexpected behavior (Zeller, 2009). Therefore, it is in the interest of the developer to "test" their software. There are many mechanisms to achieve such goal. Testing the software could be done either manually by a developer or automatically using software. Most common practice today is to test a software system as often as possible using a test suite. Such test suites are designed to maximize the amount of code they can test (code coverage) and to check if the program is producing expected results.

As is the case with the traditional software, model-driven software also requires adequate testing to guaranty it's quality. However, the classical

Email address: ali.parsai@student.uantwerpen.be (Ali Parsai)

approach to software testing is not enough to handle this task (Fleurey, Steel, and Baudry, 2004). In addition to testing the requirements of the software, there is a need to test the parts which can only be found in the model-driven software. Model transformation is one of those parts which needs special attention during testing of model-driven software. There are systematic ways of conducting such tests (France and Rumpe, 2007) which are being used today in model-driven context.

The demand for testing software, be it in traditional or model-driven context, require a certain level of quality in the test suite itself. The reality of the situation is that there are a lot of legacy systems with inadequate test suites which need improvement to become of a level which can be useful for developers. Therefore, there existed a need for a metric to find out the quality of the available test suite in a stable way. Out of this demand, an idea like mutation testing was born. Mutation testing can provide a formal method to determine the quality of a test suite by injecting intentional bugs into a system and counting the number of caught bugs.

In this article, we are going to read about the details of how mutation testing works, and how it was adapted into model-driven context. Then the setup and results of the experiment regarding this issue is going to be discussed, and a final conclusion would be presented.

2. Mutation Testing

In order to have a reliable way to determine the quality of a test suite, there should be a formal method of testing the test suite. One such method is mutation testing (DeMillo, Lipton, and Sayward, 1978). There are several studies which show that mutation testing is successful in simulating the real-life bugs a test suite might catch (Andrews, Briand, and Labiche, 2005) Just, Jalali, Inozemtseva, Ernst, Holmes, and Fraser (2014). Therefore its application in model-driven software is an interesting subject for researchers of the field.

In mutation testing, first, faulty versions of software is created by applying a single change in the system. This change is created by a mutation operator. A mutation operator is a piece of software which changes the system under test in such a way that it includes a single bug. After generating *mutants*, the test suite is run on each of them to see if any tests fail. If there is an error or failure, the mutant is regarded as killed. However, if all tests pass, it means that the bug could not be caught by the test suite and the mutant

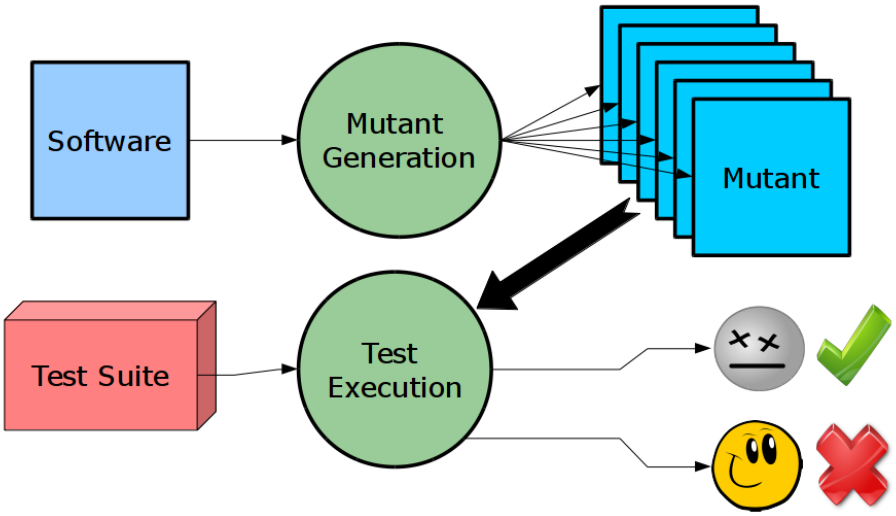

Figure 1: Mutation Testing

is regarded as survived. The final result is the number of killed mutants divided by the number of all generated mutants. This procedure can be seen in figure 1.

There are a lot of different mutation operators designed for different contexts, aiming to emulate the realistic kind of bugs in that context. Classical mutation operators effected very basic elements of the code such as the result of an expression in a conditional clause. Object-Oriented mutation operators aimed at the relationship between the different objects to introduce bugs into the system (Ma, Kwon, and Offutt, 2002). In model-driven context, these kinds of mutation operators would not prove to be useful, since any resulting model which does not correspond to the meta-model could be rejected instantly, making the final results unreliable (Mottu, Baudry, and Le Traon, 2006). Therefore this idea was adapted into model-driven context.

3. Model-Driven Mutation Testing

In order to tackle the problem of adapting mutation testing into model-driven context, we need to consider the modeling concepts which restrict the use of classical mutation operators. There are several reasons why classical mutation operators are not adequate for model-driven context. First, a faulty model usually needs to be different in a complicated way to have

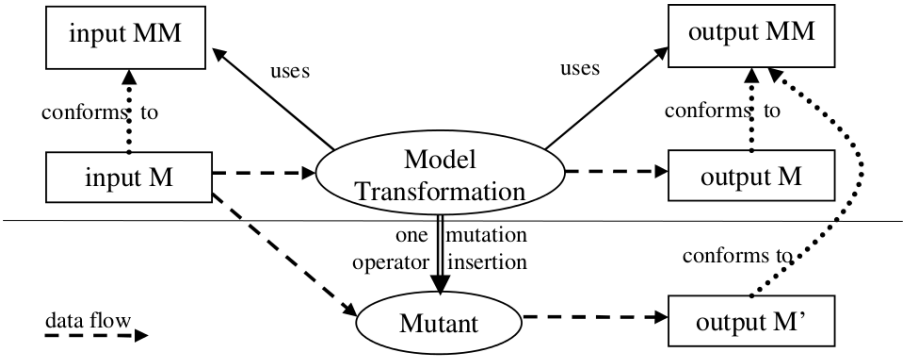

Figure 2: Model-Driven Mutation Testing

any significant effect on the outcome. This is in contrast to the purpose of classical mutation operators which is to induce a single basic fault. Second, the mutant should conform to the same meta-model as the original software; but classical mutation operators are very likely to produce a non-conforming mutant due to the lack of awareness about modeling concepts. And finally, the classical mutation operators act on the syntactic part of the software rather than the semantic part, and therefore are dependent on the language that is being used.

One particular use case of mutation testing in model-driven context is testing model transformations. In this particular use case, mutant transformation should be able to process the input models which conform to the input meta-model and should produce an output model which conforms to the output meta-model (figure 2).

To generate such mutants, we will be using mutation operators known as semantic operators which are designed specifically for this context. To read more about these operators please refer to the reading report for the project.

4. Higher-Order Transformations

To realize the goal of mutating a model transformation, we need to use a mechanism to induce a bug into the model transformation. One approach is to treat the transformation as a code entity and change its definition by changing its syntax. Another way to solve this issue is to treat the transformation as a model, and change the semantics of the model rather than working with syntax. The latter solution would provide better results since

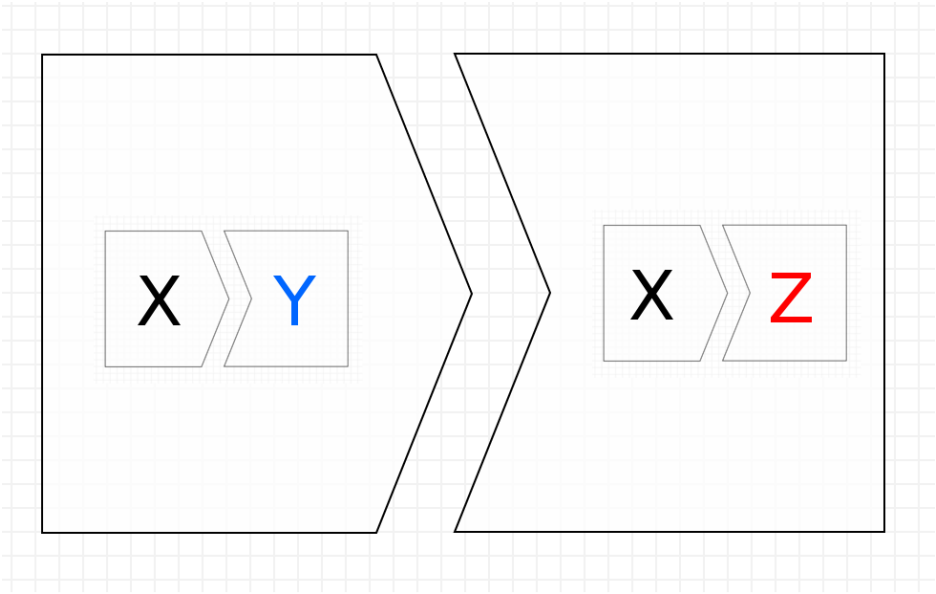

Figure 3: Higher-Order Transformation

it conforms to model-driven principles and it is free from the problems associated with syntax processing such as language dependency.

To achieve this, we need to use *higher-order transformations*. A higher-order transformation is defined as a model transformation such that its input and/or output models are themselves transformation models (Tisi, Jouault, Fraternali, Ceri, and Bzivin, 2009). A higher-order transformation can be defined to transform a correct transformation into a faulty one by introducing a change in its semantics (figure 3). Therefore this is a viable solution for creating mutation operators (Syriani, Vangheluwe, Mannadiar, Hansen, Van Mierlo, and Ergin, 2013).

5. Experiment

5.1. AToMPM

To implement the concept of mutation operators as higher-order transformations, we need a tool which supports higher-order transformations. There-

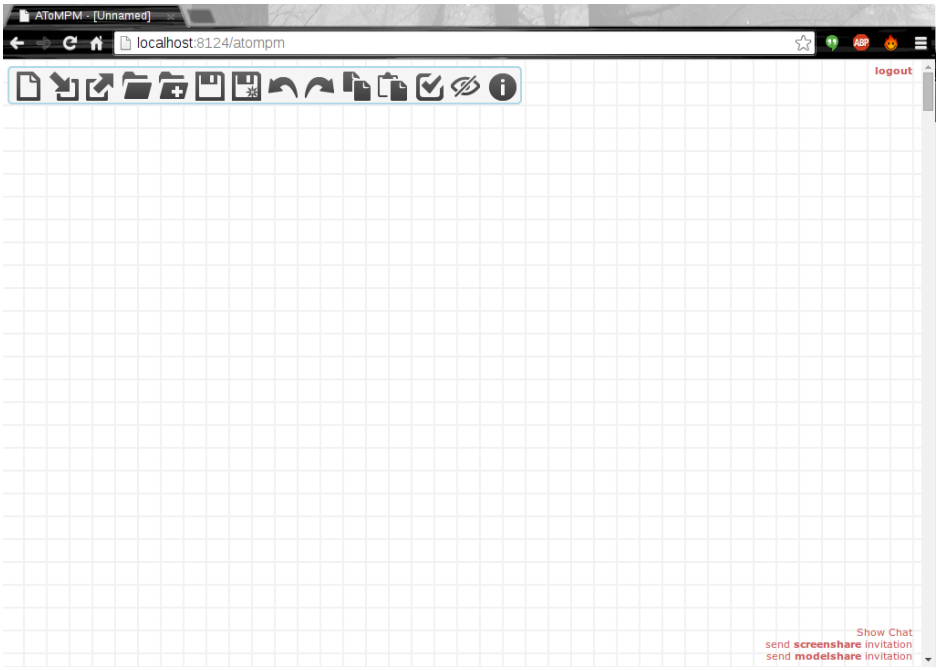

Figure 4: AToMPM Environment

fore, AToMPM[1] was selected as the tool of choice based on three reasons. First, it partially supports higher-order transformations. Second, it implements a pure approach to model-driven development by modeling everything explicitly. Third, it was the tool of choice during the instruction of the course, and therefore it was a more familiar environment compared to other tools.

5.2. RPG Game

The meta-model that was used for the implementation phase was the simple version of one developed for the assignments of the course. The meta-model can be seen in figure 5.

5.3. Implemented Operators

In total, 110 mutation operators were implemented using AToMPM. Since each operator only changes a single entity in the model, different operators were designed for left-hand side and right-hand side of the transformations.

[1] A Tool for Multi-Paradigm Modeling
`http://syriani.cs.ua.edu/atompm/atompm.htm`

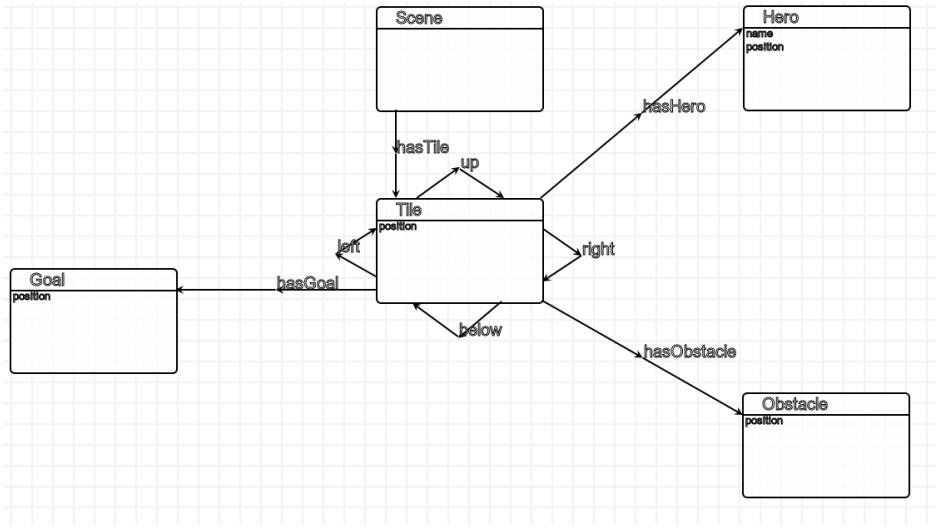

Figure 5: RPG Game Meta-Model

- **RSCC** In these operators, an association to an object is replaced by another association to the same object. Since Tile is the only class in the meta-model which has multiple associations, all the possible combinations of this operator was implemented which resulted in a total of 24 operators (12 for each side). An example of this operator can be seen in figure 6.

- **ROCC** In these operators, an association to an object is replaced by another association to another object. Implementation of this kind of operators resulted in a total of 42 operators. Considering every possibility would have resulted in 60 operators; However, considering all four associations of Tile as the replacement would create redundant operators. An example of this operator can be seen in figure 7.

- **RSMD** In these operators, an association to an object is removed to create a different transformation. Implementation of this kind of operators resulted in a total of 14 operators. An example of this operator can be seen in figure 8.

- **RSMA** In these operators, an association to an object is added to create a different transformation. Implementation of this kind of operators resulted in a total of 8 operators. Considering every possibility

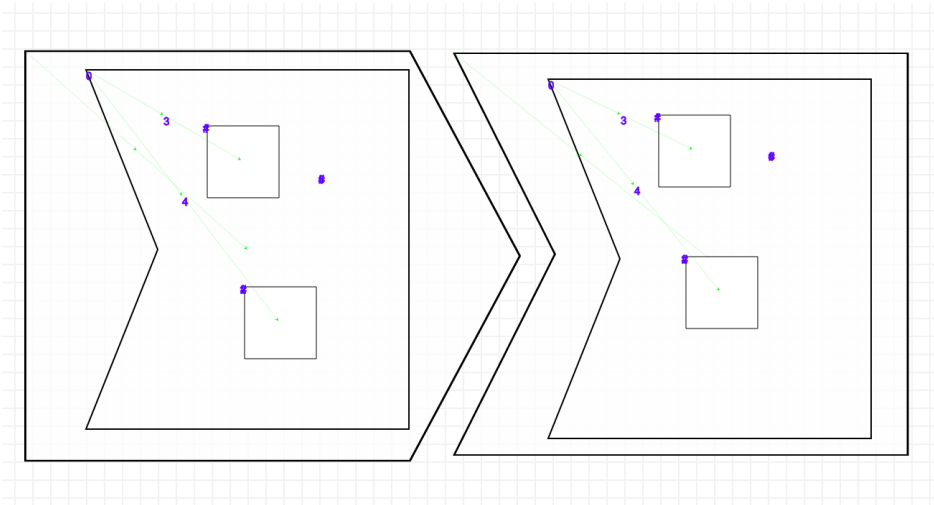

Figure 6: RSCC Operator

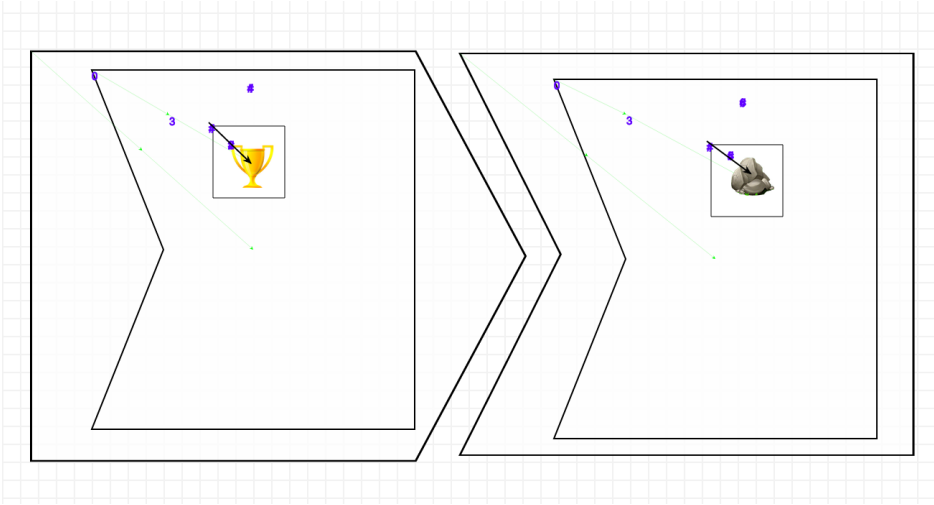

Figure 7: ROCC Operator

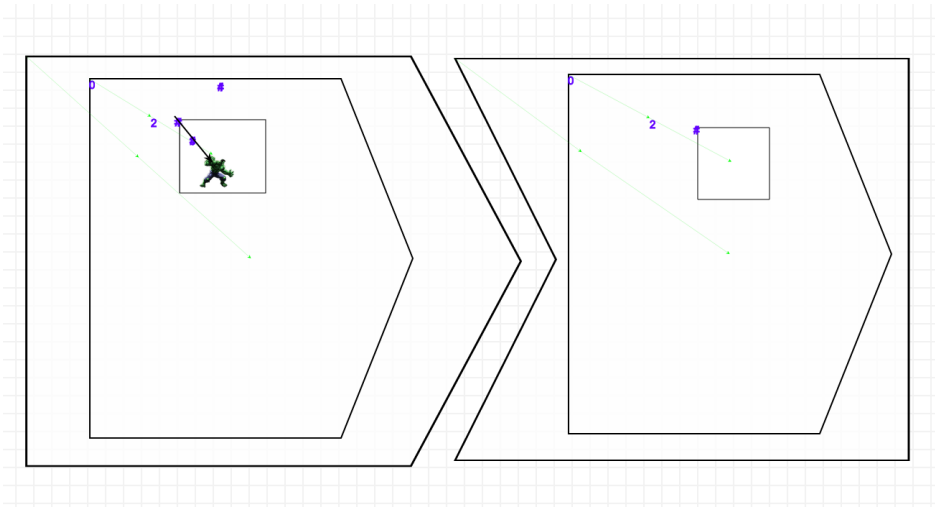

Figure 8: RSMD Operator

would have resulted in 14 operators; However, considering all four associations of Tile as the replacement would create redundant operators. An example of this operator can be seen in figure 9.

- **CACD** This operator deletes the creation of an association between two objects. A total of 14 operators were created for this kind of operator. An example of this operator can be seen in figure 10.

- **CACA** This operator adds a useless creation of an association between two objects of the output model, when the meta-model allows it. A total of 8 operators were created for this kind of operator. An example of this operator can be seen in figure 11.

5.4. Execution

In the execution phase, the operator would be run on a transformation rule, and the resulting transformation rule would be semantically wrong, even though it still conforms to the meta-model. The RSMA operator depicted in figure 9 was applied to the rule *HeroPicksGoal* seen in figure 12 to generate the faulty rule. Unfortunately, the transformation created a wrong model seen in figure 13. This can be due to either a bug in AToMPM or a bug in the transformation formalism being used. The correct expected result of the transformation could be seen in figure 14.

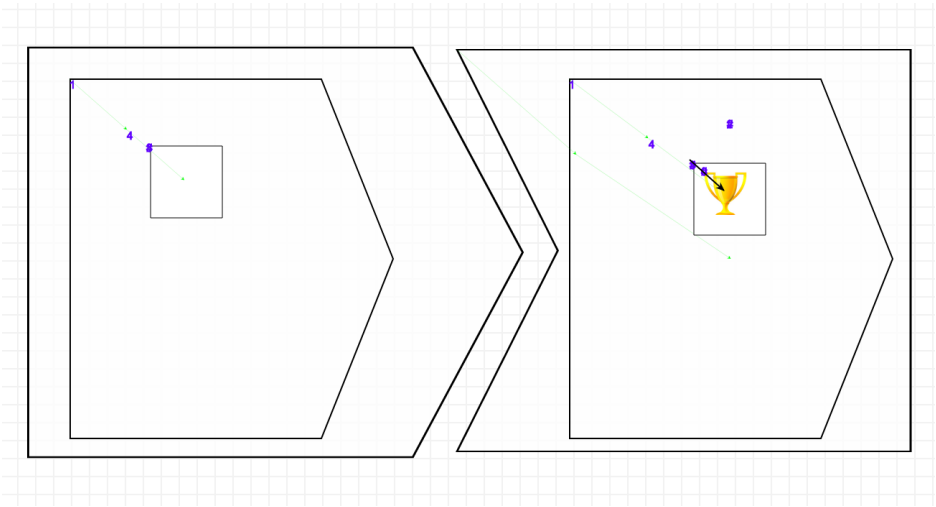

Figure 9: RSMA Operator

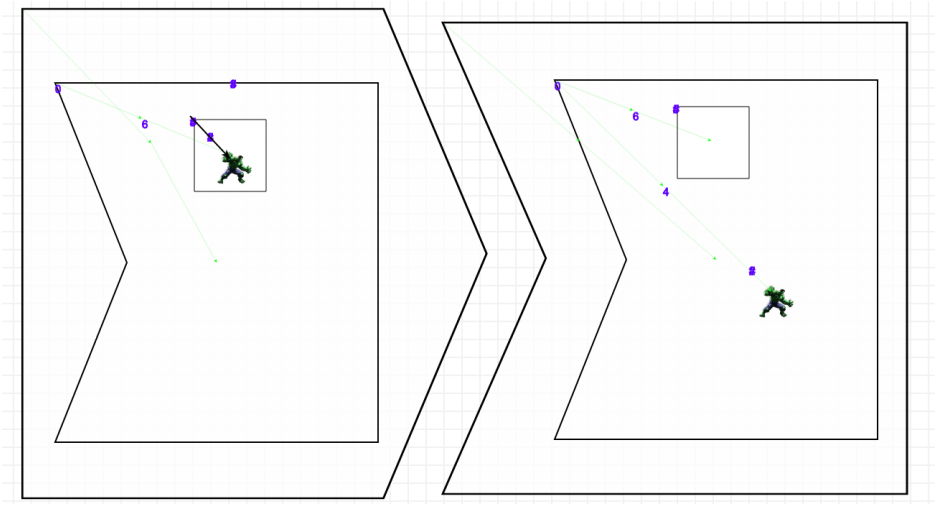

Figure 10: CACD Operator

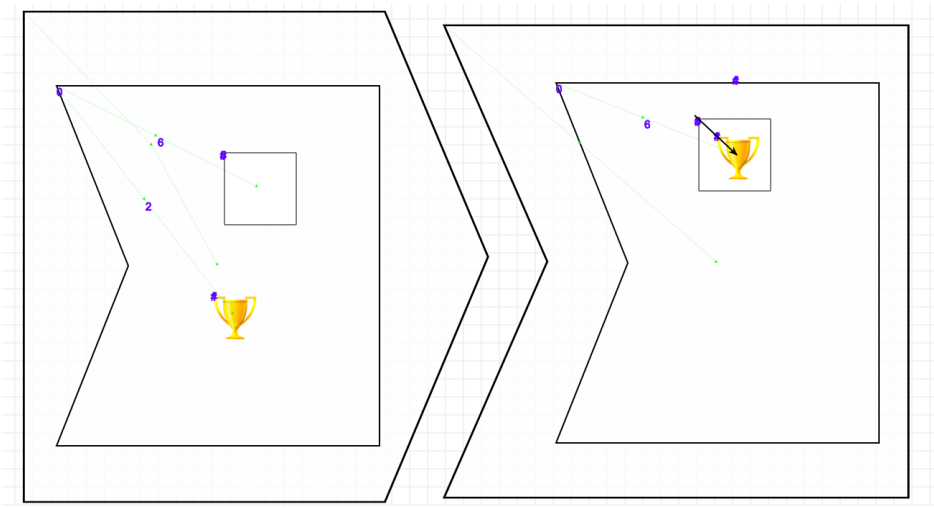

Figure 11: CACA Operator

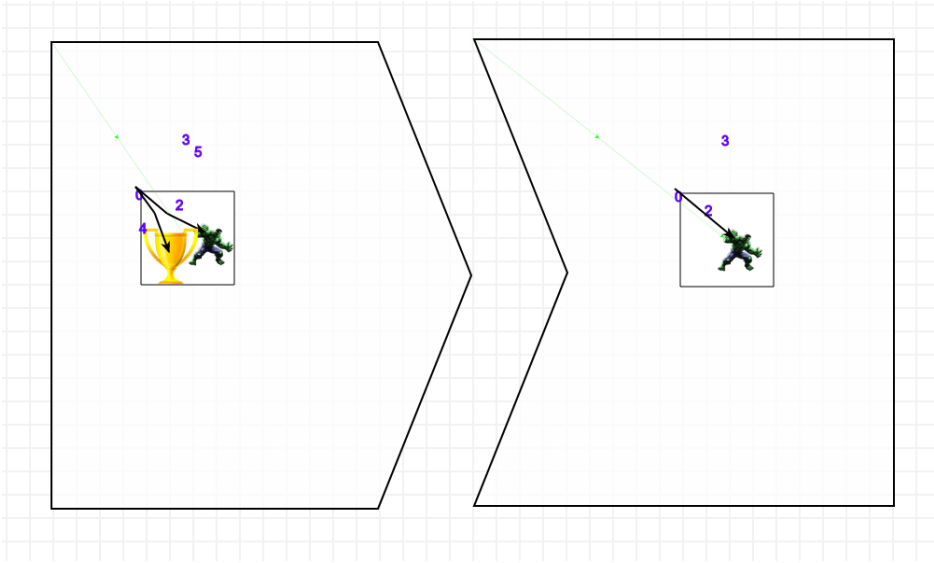

Figure 12: HeroPicksGoal Rule

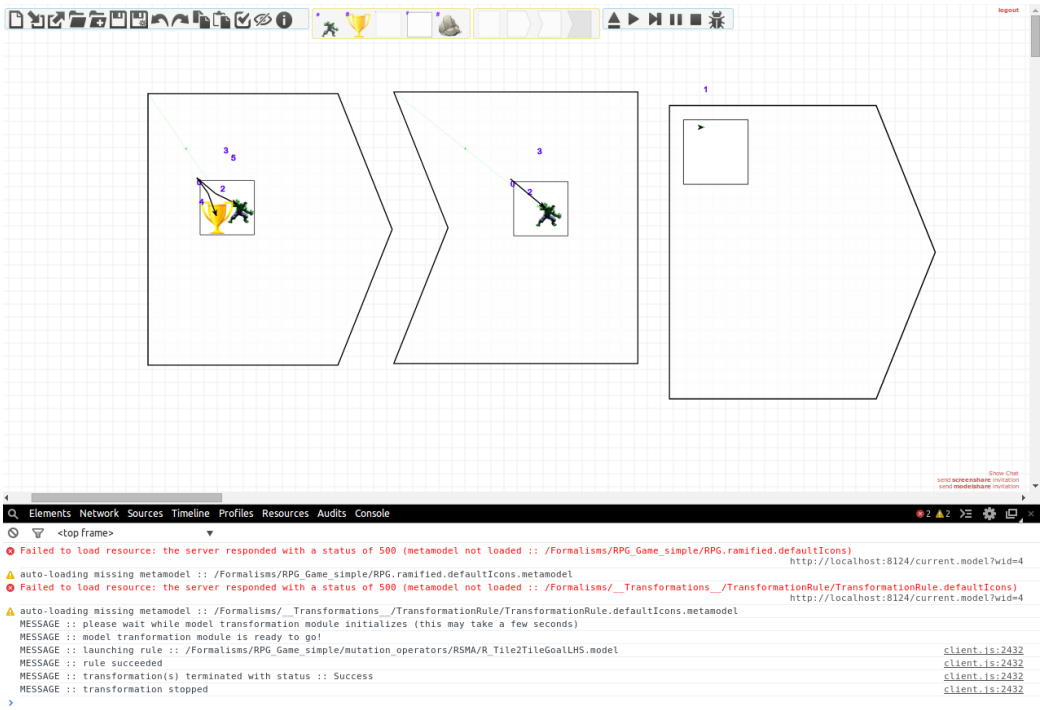

Figure 13: Problematic Transformation

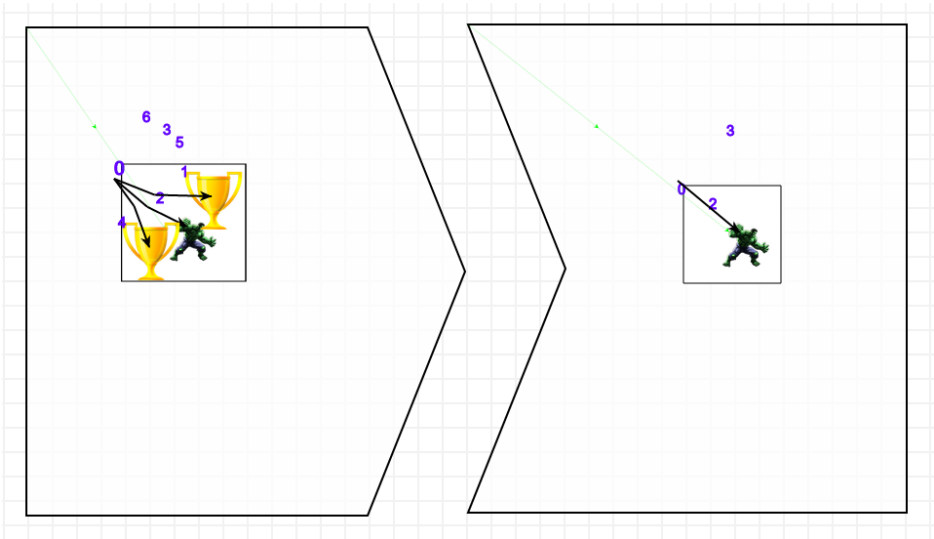

Figure 14: Expected Result of Transformation

6. Conclusion

Mutation testing in model-driven context is currently an interesting topic of research which requires rethinking the classical approaches to software testing. Many of the older methods are incompatible with this new paradigm, among which mutation testing is. Since the field is rather young compared to other fields of computer science, there are many ideas that have not been implemented yet. In this project the prospects of using higher-order transformations as a way of implementing mutation operators were explored. This method of implementation has several advantages over older methods such as conformity to pure model-driven philosophy, and awareness of the semantics of the model. AToMPM was shown to be capable of creating such operators. It was also shown that it is feasible to expect that applying of such operators on models be supported in AToMPM. The failures along the way did not happen because of a lack of features, but probably because of trivial bugs in the tool.

References

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

DeMillo, R. A., Lipton, R. J., Sayward, F. G., Apr. 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–41. URL http://dx.doi.org/10.1109/C-M.1978.218136

Fleurey, F., Steel, J., Baudry, B., Nov 2004. Validation in model-driven engineering: testing model transformations. In: Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on. pp. 29–40.

France, R., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. FOSE '07. IEEE Computer Society, Washington, DC, USA, pp. 37–54. URL http://dx.doi.org/10.1109/FOSE.2007.14

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? Tech. Rep. UW-CSE-14-02-02, University of Washington.

Ma, Y.-S., Kwon, Y.-R., Offutt, J., 2002. Inter-class mutation operators for java. In: Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on. pp. 352–363.

Mottu, J.-M., Baudry, B., Le Traon, Y., 2006. Mutation analysis testing for model transformations. In: Rensink, A., Warmer, J. (Eds.), Model Driven Architecture Foundations and Applications. Vol. 4066 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 376–390. URL http://dx.doi.org/10.1007/11787044_28

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

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bzivin, J., 2009. On the use of higher-order model transformations. In: Paige, R., Hartman, A., Rensink, A. (Eds.), Model Driven Architecture - Foundations and Applications. Vol.

5562 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 18–33.
URL http://dx.doi.org/10.1007/978-3-642-02674-4_3

Zeller, A., 2009. Why Programs Fail: A Guide to Systematic Debugging, 2nd Edition. Morgan Kaufmann, Burlington, MA.
URL http://www.sciencedirect.com/science/book/9780123745156