Mutation-based Testing of Rule-based Model Transformations Using Higher-order Transformations

Presented by: Ali Parsai

Supervisor: Prof. Vangheluwe

Model-driven Engineering 2013-2014

University of Antwerp

Software Testing

- Testing is required to assure quality
- Manual testing is not an option
- Automatic test-suites do the job

Test-suite quality

- Adequate coverage
- Ability to catch common bugs

Quis custodiet ipsos custodes?

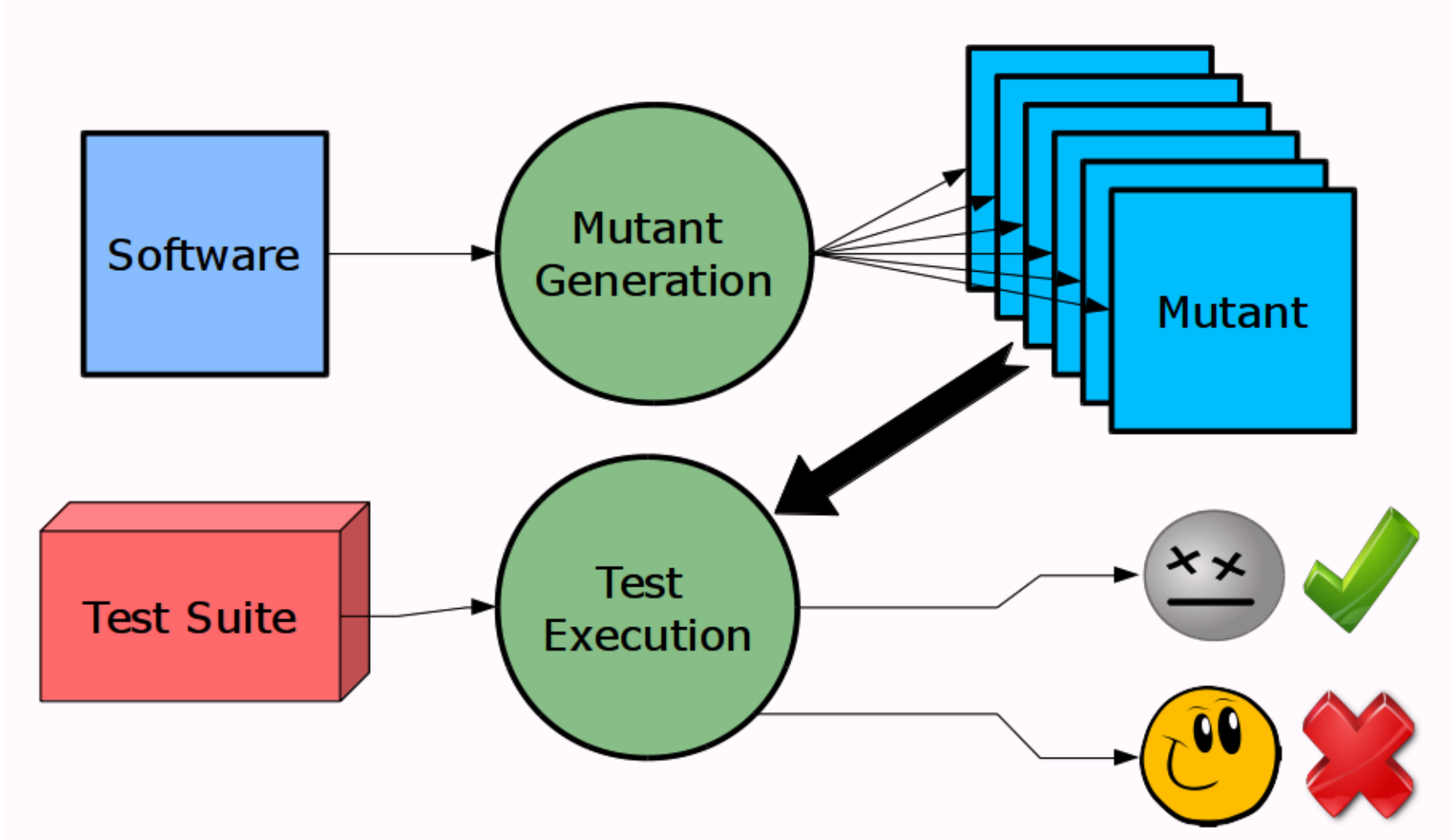
Socrates

Mutation Testing*

- Repeatable, scientific approach
- Simulates realistic situations*
- Quantifiable, tangible results

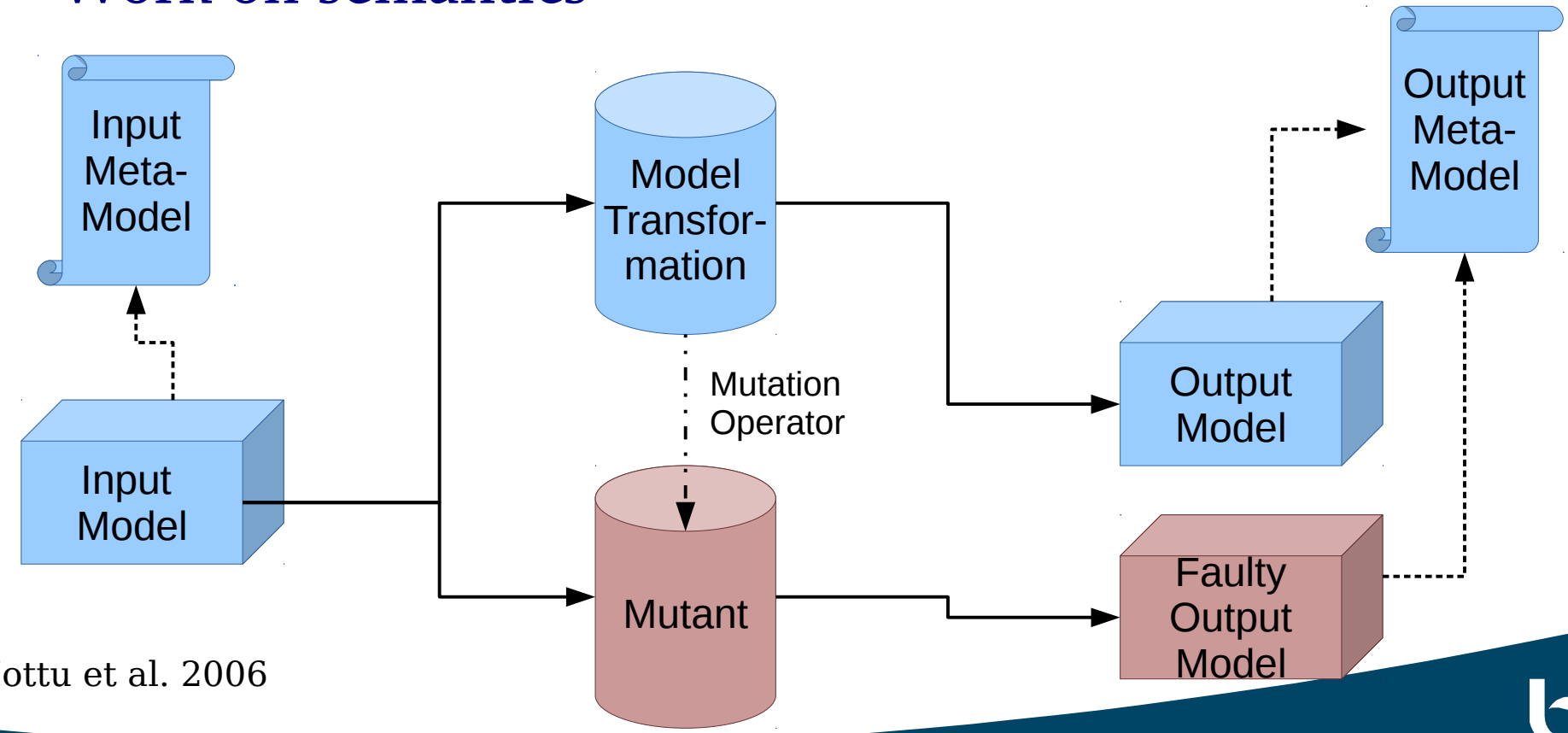
* DeMillo et al. 1978
* Just et al. 2014

Mutation Testing



Mutation Testing of Model Transformations

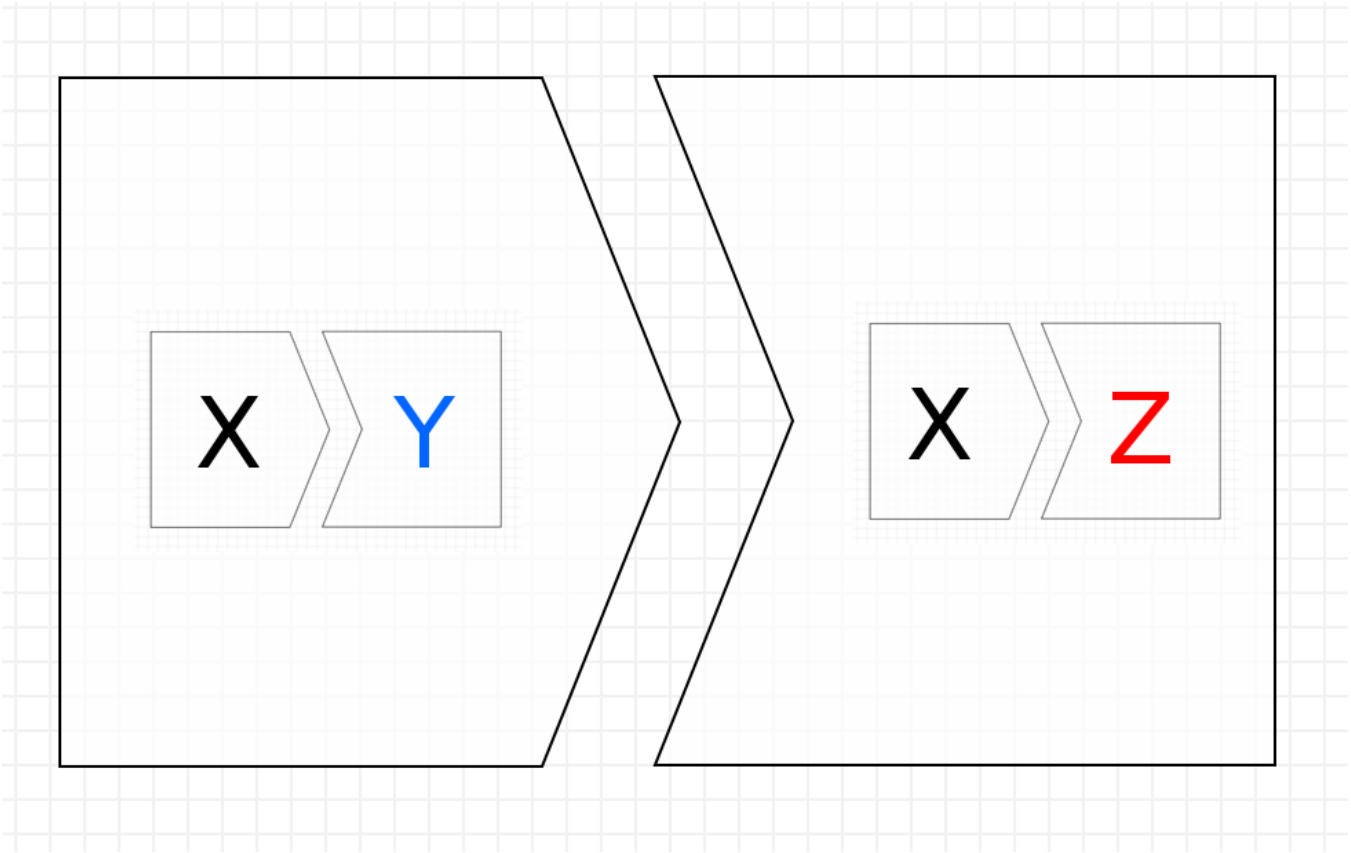
- Requires complicated change
- Mutants compatible with meta-model
- Work on semantics*



* Mottu et al. 2006

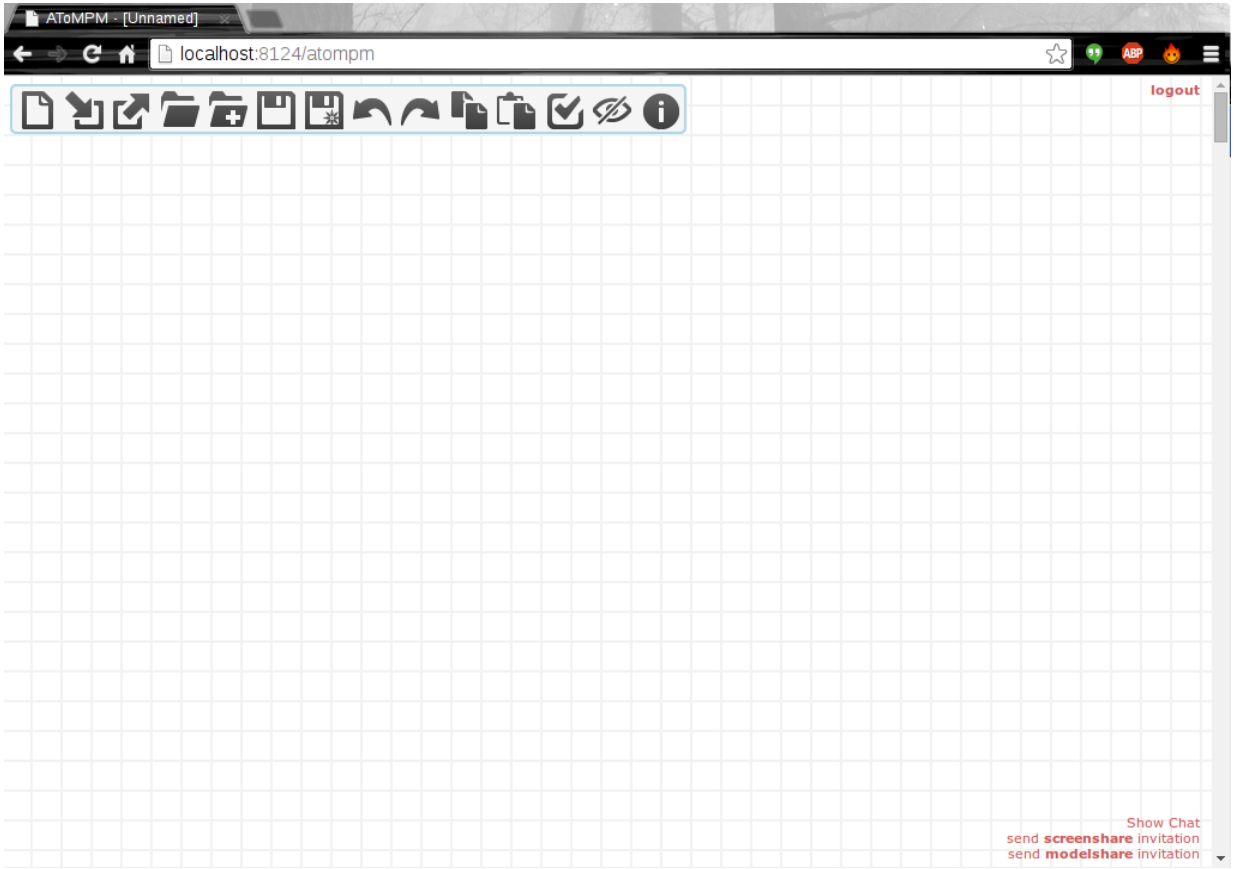
Higher-Order Transformations

- Mutation operator as a higher-order transformation



Experiment

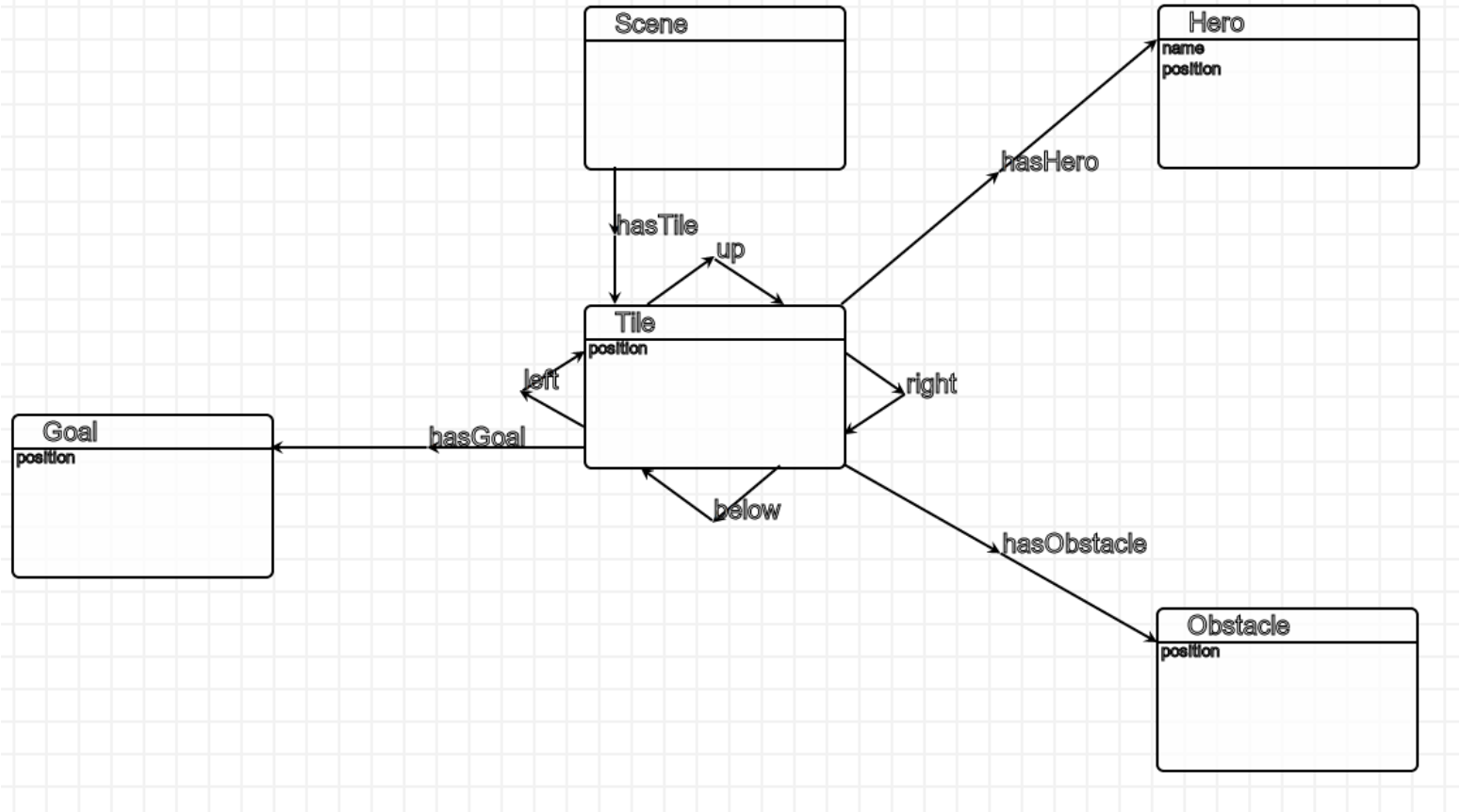
- Tool: AToMPM*



* Syriani et al. 2013

Experiment

- Meta-model: RPG Game



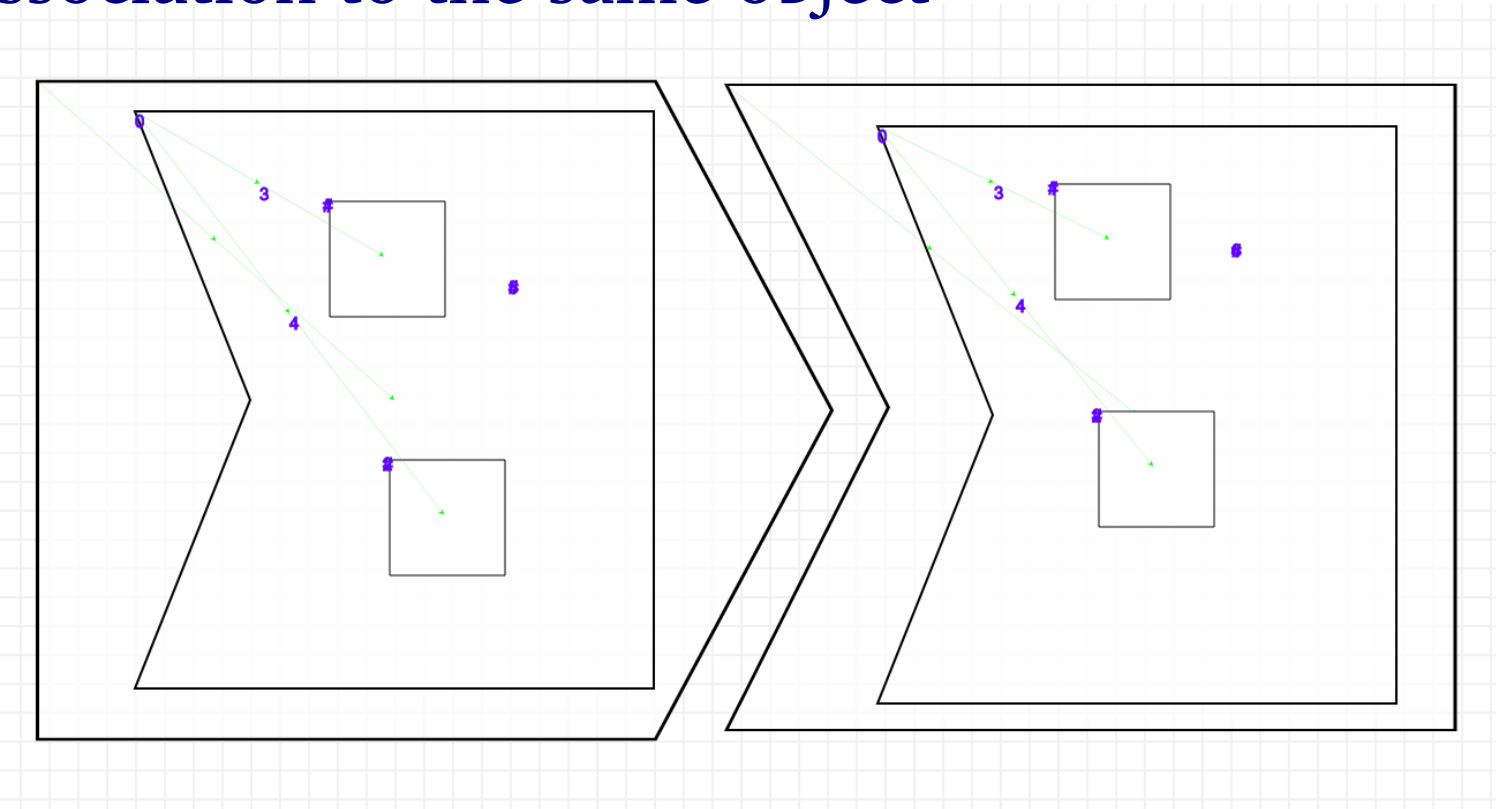
Mutation Operators

- 110 in total
- Using a double ramified version of RPG Game
- Using a ramified version of TransformationRule

- RSCC: Relation to Same Class Change
- ROCC: Relation to Other Class Change
- RSMD: Relation Sequence Modification with Deletion
- RSMA: Relation Sequence Modification with Addition
- CACD: Classes Association Creation Deletion
- CACA: Classes Association Creation Addition

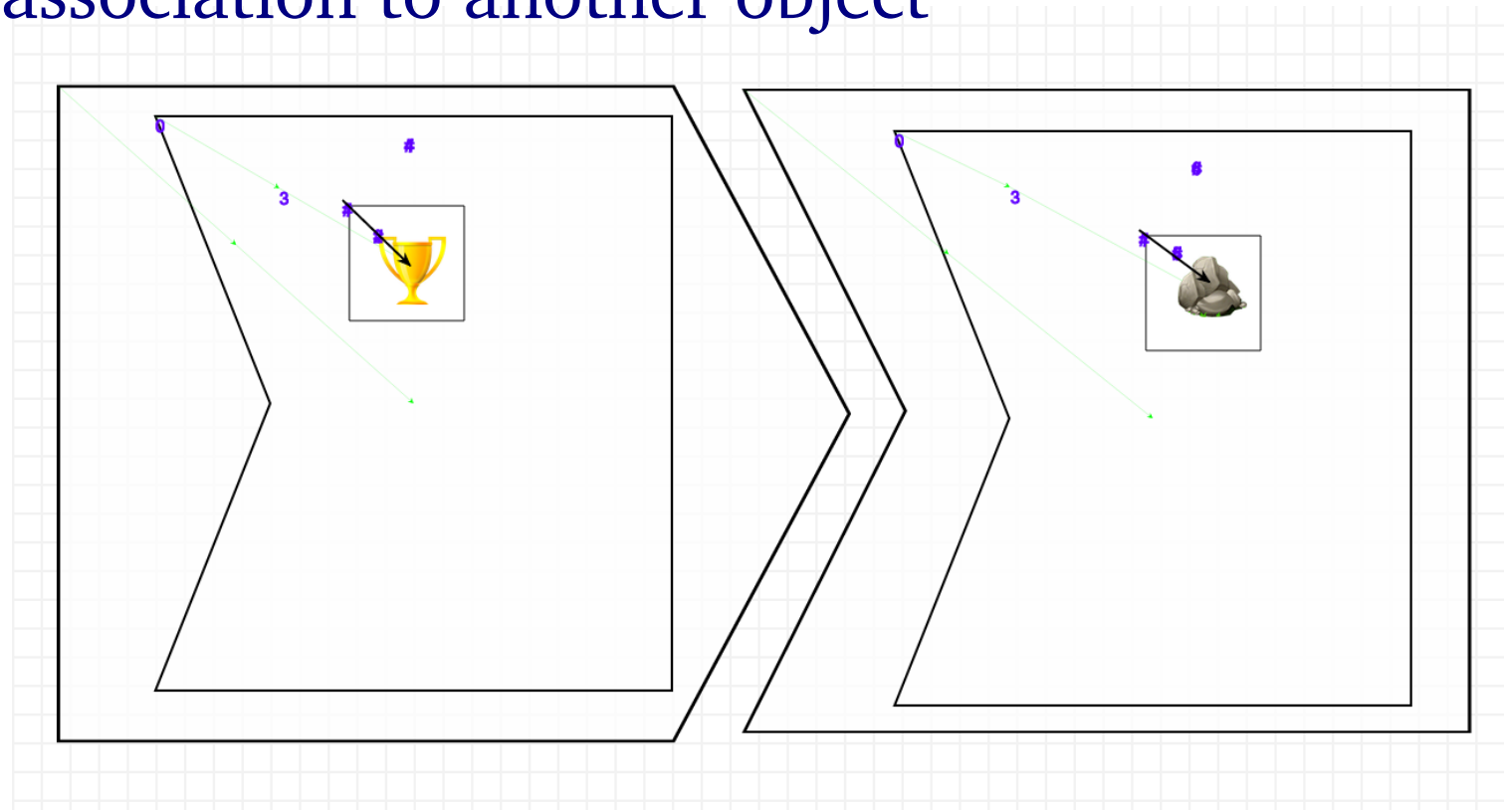
Relation to Same Class Change

- 24 in total
- An association to an object is replaced by another association to the same object



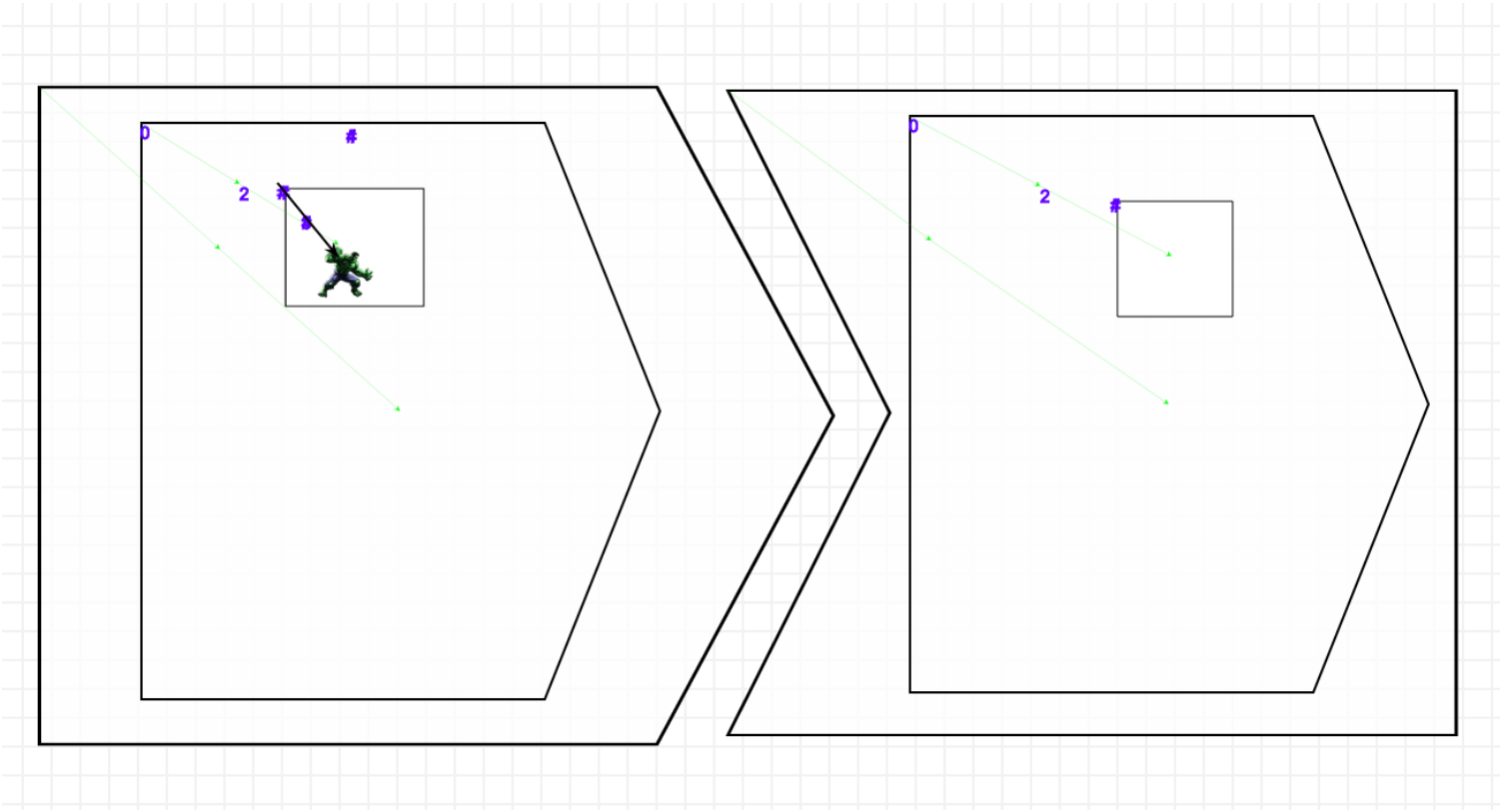
Relation to Other Class Change

- 42 in total
- An association to an object is replaced by another association to another object



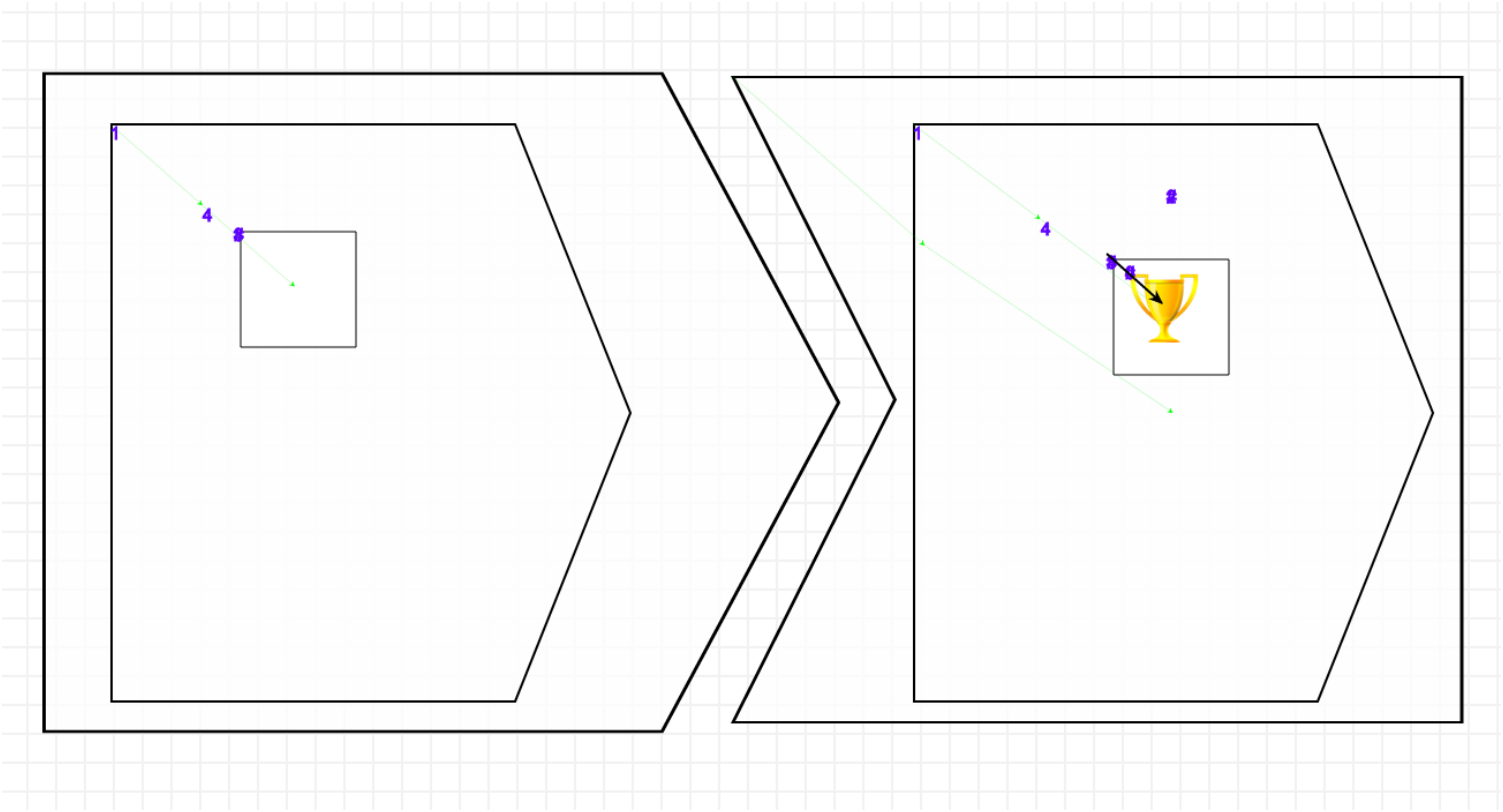
Relation Sequence Modification with Deletion

- 14 in total
- An association to an object is removed



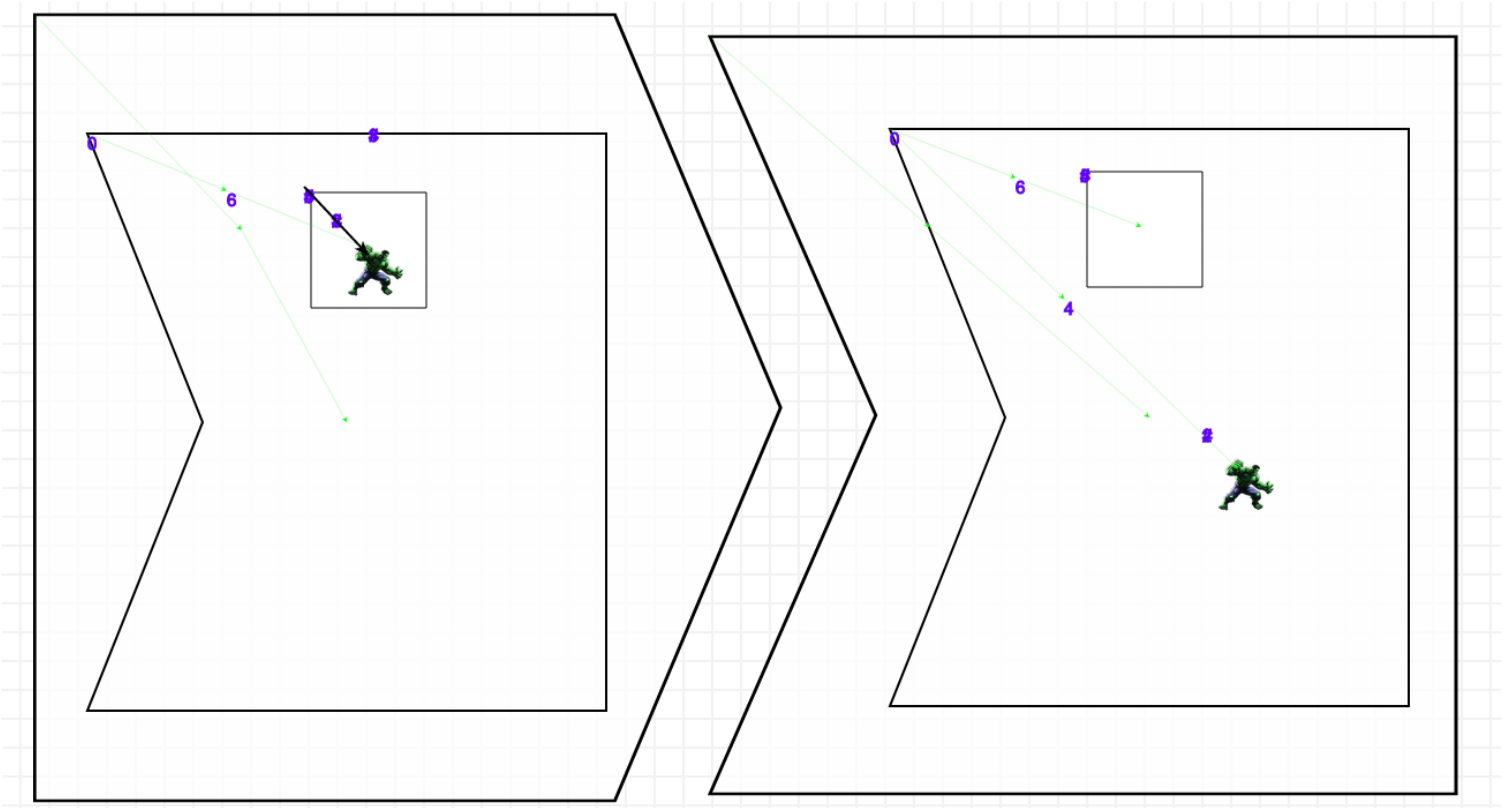
Relation Sequence Modification with Addition

- 8 in total
- An association to an object is added



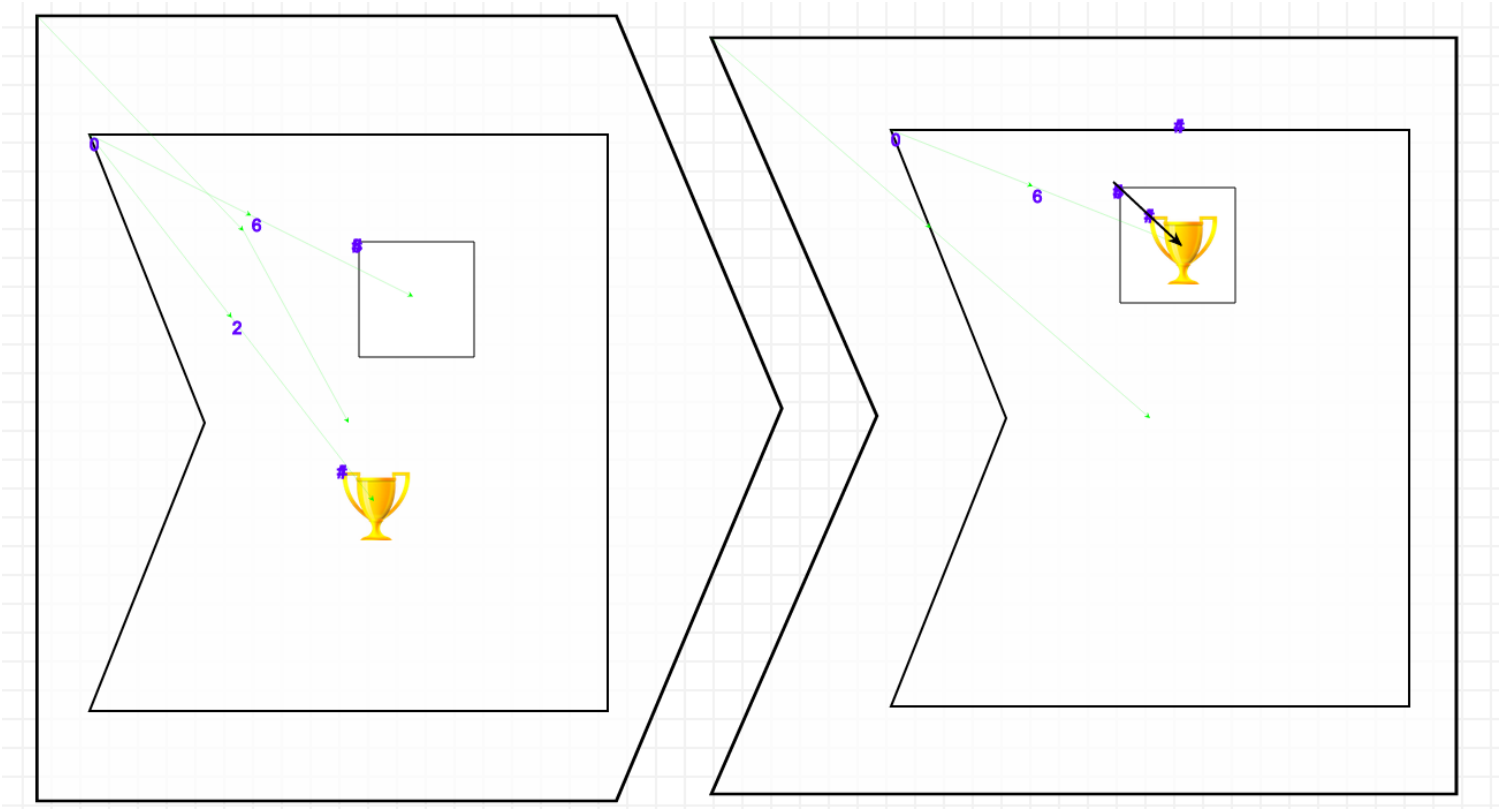
Classes Association Creation Deletion

- 14 in total
- An association between two objects is deleted



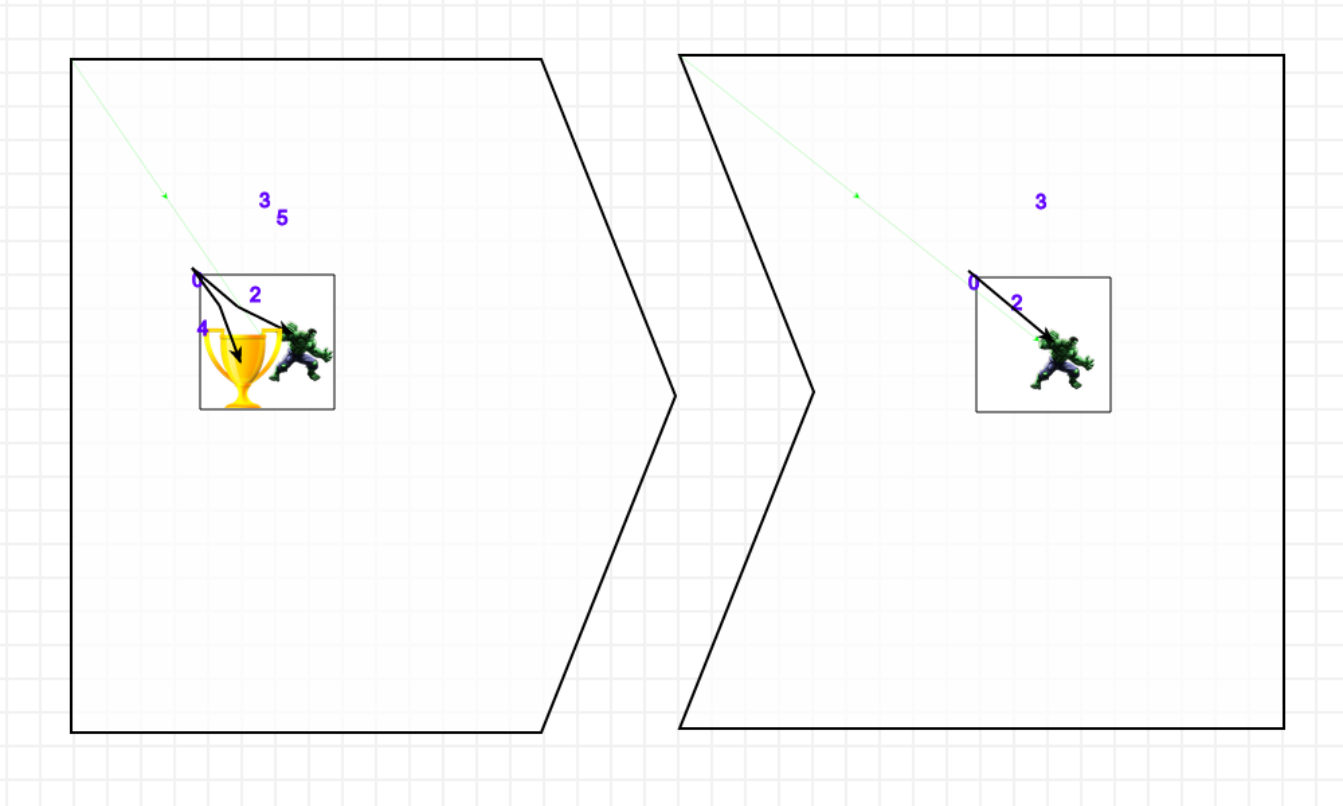
Classes Association Creation Addition

- 8 in total
- An association between two objects is added



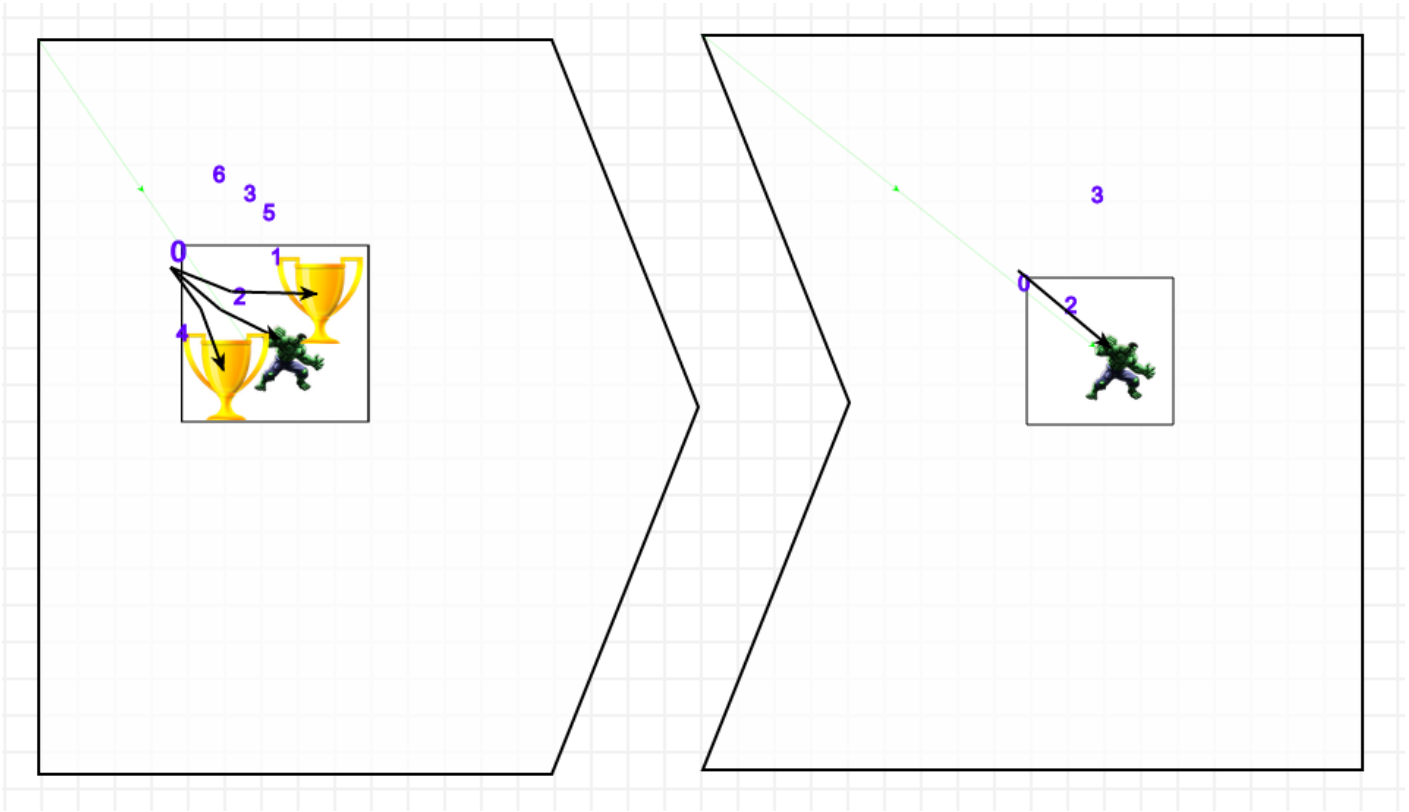
Execution

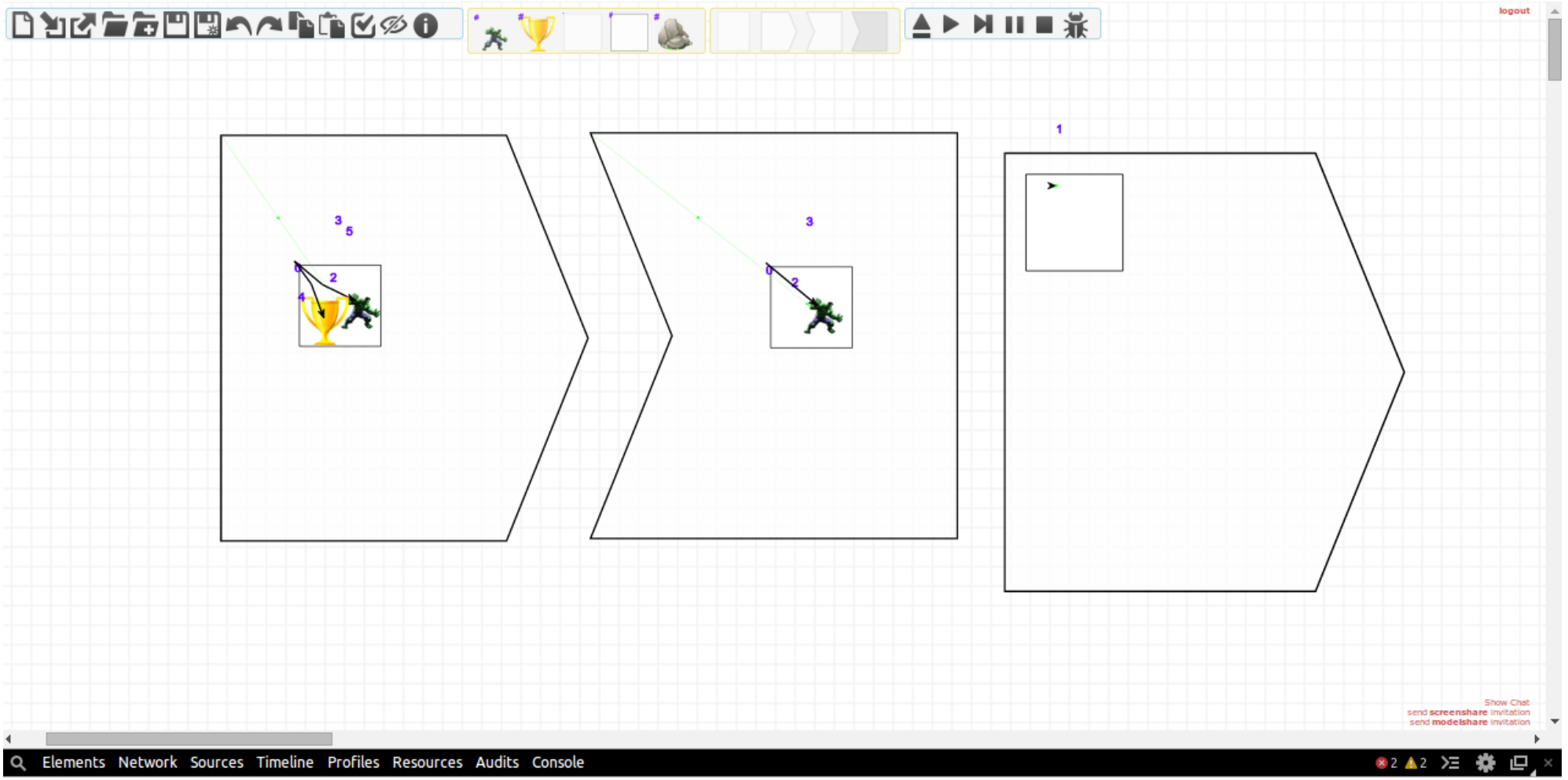
- Transformation Rule: HeroPicksGoal
- Mutation Operator Type: RSMA



Execution

- Expected Result:





Elements Network Sources Timeline Profiles Resources Audits Console

<top frame>

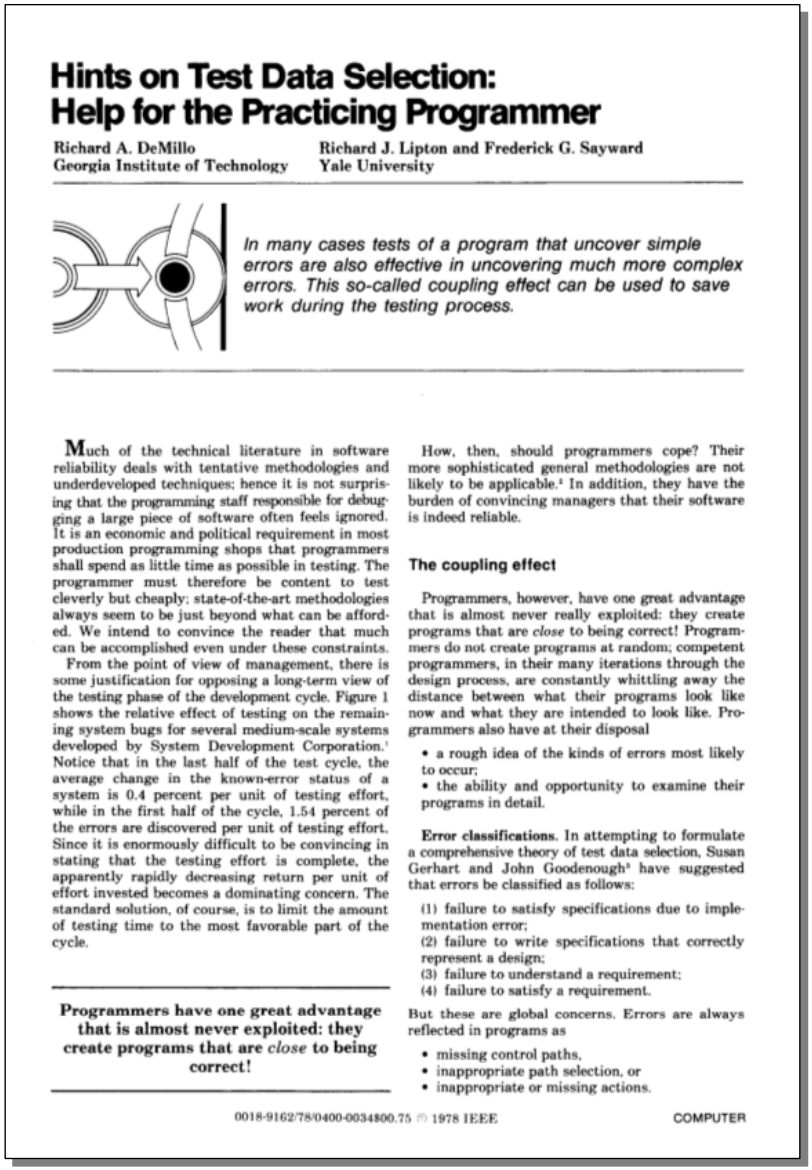
```
Failed to load resource: the server responded with a status of 500 (metamodel not loaded :: /Formalisms/RPG_Game_simple/RPG.ramified.defaultIcons)
                                                                                                    http://localhost:8124/current.model?wid=4
auto-loading missing metamodel :: /Formalisms/RPG_Game_simple/RPG.ramified.defaultIcons.metamodel
Failed to load resource: the server responded with a status of 500 (metamodel not loaded :: /Formalisms/__Transformations__/TransformationRule/TransformationRule.defaultIcons)
                                                                                                    http://localhost:8124/current.model?wid=4
auto-loading missing metamodel :: /Formalisms/__Transformations__/TransformationRule/TransformationRule.defaultIcons.metamodel
MESSAGE :: please wait while model transformation module initializes (this may take a few seconds)
MESSAGE :: model tranformation module is ready to go!
MESSAGE :: launching rule :: /Formalisms/RPG_Game_simple/mutation_operators/RSMA/R_Tile2TileGoalLHS.model        client.js:2432
MESSAGE :: rule succeeded                                                                                      client.js:2432
MESSAGE :: transformation(s) terminated with status :: Success                                                 client.js:2432
MESSAGE :: transformation stopped                                                                              client.js:2432
>
```

Conclusion

- Mutation testing can be adapted to model-driven context
- Mutation operators can be defined as higher-order transformations
- AToMPM can implement these operators

References

- 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



Hints on Test Data Selection: Help for the Practicing Programmer

Richard A. DeMillo
Georgia Institute of Technology

Richard J. Lipton and Frederick G. Sayward
Yale University

In many cases tests of a program that uncover simple errors are also effective in uncovering much more complex errors. This so-called coupling effect can be used to save work during the testing process.

Much of the technical literature in software reliability deals with tentative methodologies and underdeveloped techniques; hence it is not surprising that the programming staff responsible for debugging a large piece of software often feels ignored. It is an economic and political requirement in most production programming shops that programmers shall spend as little time as possible in testing. The programmer must therefore be content to test cleverly but cheaply; state-of-the-art methodologies always seem to be just beyond what can be afforded. We intend to convince the reader that much can be accomplished even under these constraints.

From the point of view of management, there is some justification for opposing a long-term view of the testing phase of the development cycle. Figure 1 shows the relative effect of testing on the remaining system bugs for several medium-scale systems developed by System Development Corporation.[1] Notice that in the last half of the test cycle, the average change in the known-error status of a system is 0.4 percent per unit of testing effort, while in the first half of the cycle, 1.54 percent of the errors are discovered per unit of testing effort. Since it is enormously difficult to be convincing in stating that the testing effort is complete, the apparently rapidly decreasing return per unit of effort invested becomes a dominating concern. The standard solution, of course, is to limit the amount of testing time to the most favorable part of the cycle.

Programmers have one great advantage that is almost never exploited: they create programs that are *close* to being correct!

How, then, should programmers cope? Their more sophisticated general methodologies are not likely to be applicable.[4] In addition, they have the burden of convincing managers that their software is indeed reliable.

The coupling effect

Programmers, however, have one great advantage that is almost never really exploited: they create programs that are *close* to being correct! Programmers do not create programs at random; competent programmers, in their many iterations through the design process, are constantly whittling away the distance between what their programs look like now and what they are intended to look like. Programmers also have at their disposal

- a rough idea of the kinds of errors most likely to occur;
- the ability and opportunity to examine their programs in detail.

Error classifications. In attempting to formulate a comprehensive theory of test data selection, Susan Gerhart and John Goodenough[5] have suggested that errors be classified as follows:

(1) failure to satisfy specifications due to implementation error;
(2) failure to write specifications that correctly represent a design;
(3) failure to understand a requirement;
(4) failure to satisfy a requirement.

But these are global concerns. Errors are always reflected in programs as

- missing control paths,
- inappropriate path selection, or
- inappropriate or missing actions.

0018-9162/78/0400-0034$00.75

References

- 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

Are mutants a valid substitute for real faults in software testing?

René Just[1], Darioush Jalali[1], Laura Inozemtseva[2], Michael D. Ernst[1], Reid Holmes[2], and Gordon Fraser[3]
[1]University of Washington, Seattle, WA, USA
[2]University of Waterloo, Waterloo, ON, Canada
[3]University of Sheffield, Sheffield, UK

ABSTRACT

A good test suite is one that detects real faults. Because the set of faults in a program is unknowable, this definition is not useful to practitioners who are creating test suites nor to researchers who are creating and evaluating tools that generate test suites. In place of real faults, testing research often uses mutants, which are artificial faults — each one a simple syntactic variation — that are systematically seeded throughout the program under test. Mutation testing is appealing because large numbers of mutants can be automatically generated and used as a proxy for real faults.

Unfortunately, there is little experimental evidence to support the use of mutants as a proxy for real faults. This paper investigates whether mutants are indeed a valid substitute for real faults — that is, whether a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed.

Our experiments used 357 real faults in 5 open-source applications totalling 321,000 lines of source code. Furthermore, our experiments used both developer-written and generated test suites. We found a statistically significant correlation between mutant detection and real fault detection, even when controlling for code coverage.

1. INTRODUCTION

Both industrial software developers and software engineering researchers are interested in measuring test suite quality: developers want to know if their suites have a good chance of detecting faults, while researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown, so a proxy measurement must be used instead.

A well-established proxy for test quality in testing research is the mutation score, which measures a test suite's ability to distinguish a program under test (*original version*) from many small syntactic variations, called *mutants*. The *mutation score* is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, based on *mutation operators*. Examples of such mutation operators are replacement of arithmetic operators (e.g., $x+y \mapsto x-y$), modification of branch conditions, or deletion of statements. A test suite that can detect (or *kill*) more mutants — that is, it has a higher mutation score — is considered to be a better suite than one that detects fewer mutants.

This measurement is often used in software testing and debugging research. More concretely, mutation analysis is commonly used in the following use cases (e.g., [3, 13, 25, 26]):

1. *Test suite augmentation and generation*
   A test suite T is only augmented with a test t if this test increases the mutation score of T. Likewise, a mutation-based test generation approach generates and optimizes a test suite towards its mutation score based on the assumption that a higher mutation score indicates a better test suite.
2. *Test suite selection and evaluation*
   Suppose we have two unrelated test suites T_1 and T_2 that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score. Generally in the context of test suite evaluation, a test suite that has a higher mutation score is assumed to be more effective with respect to real faults.
3. *Test suite minimization*
   In the context of test suite minimization, a test suite T is reduced to $T \setminus \{t\}$ for every test $t \in T$ for which the reduction does not decrease the mutation score of T.
4. *Fault localization*
   A fault localization technique that precisely identifies a mutation location as the root cause of this artificial fault is assumed to be equally effective for real faults.

These uses of mutation analysis rely on the fundamental assumption that mutants are a valid substitute for real faults. Two test suites with the same mutation score are assumed to be equally effective — that is, they are assumed to have the same real fault detection capability. However, there is surprisingly little experimental evidence supporting this assumption, as discussed in greater detail in Section 4.

To the best of our knowledge, only two previous studies have explored the correlation between mutants and real faults [1,5]. These studies used small programs — the largest had only 5,905 LOC. In addition, one study investigated only 12 real faults [5] while the other examined 38 real faults [1]. Due to the lack of real faults, the latter study also used hand-seeded faults for 7 out of its 8 subject programs. However, it is not clear that these hand-seeded faults are equivalent to real inadvertently-introduced faults.

Besides, prior research also neglected the effect of structural code coverage when studying the correlation between mutant detection and real fault detection. A higher mutation score could simply be caused by a higher code coverage. Therefore, it is not clear how mutant detection is correlated with real fault detection independently of code coverage — that is, whether this correlation exists even if coverage is controlled for.

This paper extends previous work and explores the relationship between mutants and real faults using 5 large Java programs and

References

- 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

Mutation Analysis Testing for Model Transformations

Jean-Marie Mottu[1], Benoit Baudry[1], and Yves Le Traon[2]

[1] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{jean-marie.mottu, bbaudry}@irisa.fr
[2] France Télécom R&D, 2 av. Pierre Marzin 22307 Lannion Cedex, France
yves.letraon@francetelecom.com

Abstract. In MDE, model transformations should be efficiently tested so that it may be used and reused safely. Mutation analysis is an efficient technique to evaluate the quality of test data, and has been extensively studied both for procedural and object-oriented languages. In this paper, we study how it can be adapted to model oriented programming. Since no model transformation language has been widely accepted today, we propose generic fault models that are related to the model transformation process. First, we identify abstract operations that constitute this process: model navigation, model's elements filtering, output model creation and input model modification. Then, we propose a set of specific mutation operators which are directly inspired from these operations. We believe that these operators are meaningful since a large part of the errors in a transformation are due to the manipulation of complex models regardless of the concrete implementation language.

1 Introduction

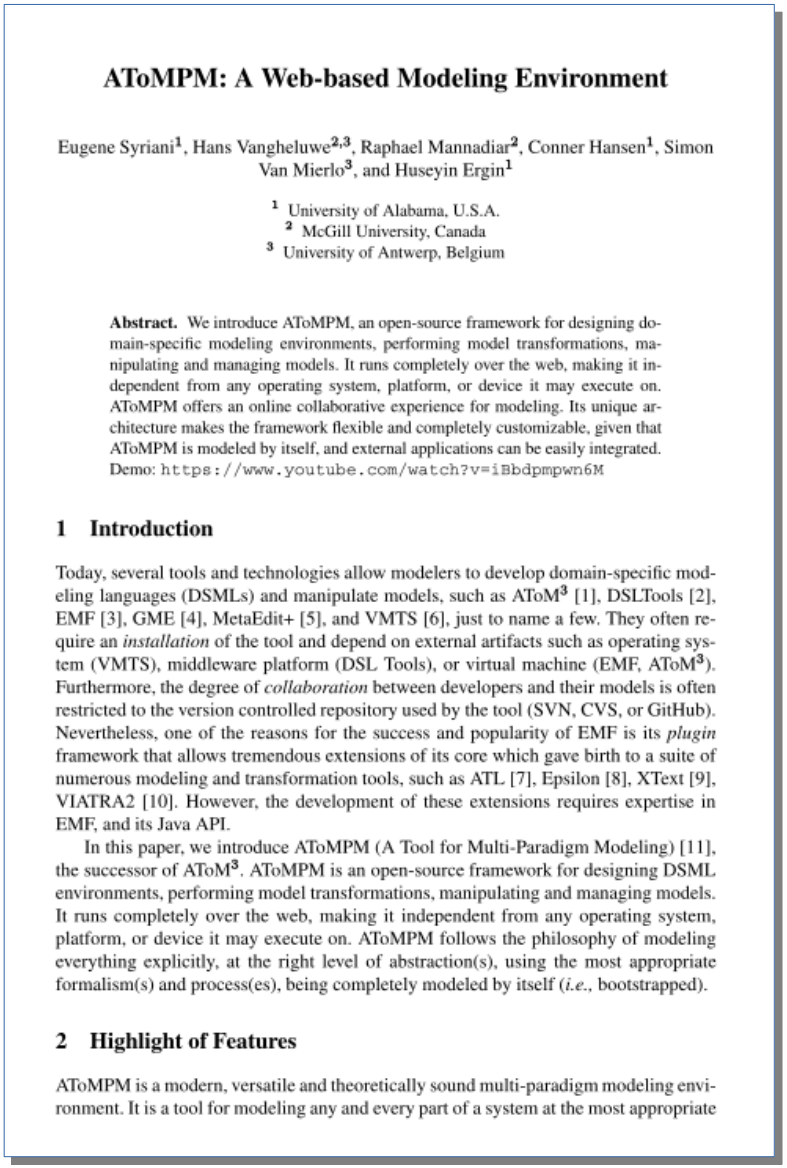
Validation refers to a process that aims at increasing our confidence that software meets its requirements. It usually relies on a combination of reasoning and testing, and encompasses unit, integration, and acceptance testing. Testing is thus a key aspect of software development, because of its cost and impact on final product reliability.

In the case of model-driven development, classical views on testing and their associated testing models are not well-suited to the significant changes this software paradigm has induced to the development process. The standardization of a model transformation language (QVT) reveals the need of a systematic way for specifying and implementing the model transformations. However, as for any other program, faults may occur in a model transformation program which must be detected through testing. Programming a model transformation is a very specific task which implies operations a classical programmer does not usually manipulate, such as navigating the input/output metamodels or filtering model elements in collections. If a skilled programmer of a model transformation can still introduce classical faults in the program, specific faults appear. These specific faults are more at a semantic level than classical programming faults. For instance, the programmer may have navigated a wrong association from class A to class B, thus manipulating class incorrect instances of the expected type. He may also be wrong in the criteria used to select some class instances (e.g. selecting all classes while he should have selected only persistent ones). Such faults are related to new fault categories we introduce in this paper.

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 376 – 390, 2006.

References

- 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



AToMPM: A Web-based Modeling Environment

Eugene Syriani[1], Hans Vangheluwe[2,3], Raphael Mannadiar[2], Conner Hansen[1], Simon Van Mierlo[3], and Huseyin Ergin[1]

[1] University of Alabama, U.S.A.
[2] McGill University, Canada
[3] University of Antwerp, Belgium

Abstract. We introduce AToMPM, an open-source framework for designing domain-specific modeling environments, performing model transformations, 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 offers an online collaborative experience for modeling. Its unique architecture makes the framework flexible and completely customizable, given that AToMPM is modeled by itself, and external applications can be easily integrated. Demo: `https://www.youtube.com/watch?v=iBbdpmpwn6M`

1 Introduction

Today, several tools and technologies allow modelers to develop domain-specific modeling languages (DSMLs) and manipulate models, such as AToM^3 [1], DSLTools [2], EMF [3], GME [4], MetaEdit+ [5], and VMTS [6], just to name a few. They often require an *installation* of the tool and depend on external artifacts such as operating system (VMTS), middleware platform (DSL Tools), or virtual machine (EMF, AToM^3). Furthermore, the degree of *collaboration* between developers and their models is often restricted to the version controlled repository used by the tool (SVN, CVS, or GitHub). Nevertheless, one of the reasons for the success and popularity of EMF is its *plugin* framework that allows tremendous extensions of its core which gave birth to a suite of numerous modeling and transformation tools, such as ATL [7], Epsilon [8], XText [9], VIATRA2 [10]. However, the development of these extensions requires expertise in EMF, and its Java API.

In this paper, we introduce AToMPM (A Tool for Multi-Paradigm Modeling) [11], the successor of AToM^3. AToMPM is an open-source framework for designing DSML environments, performing model transformations, 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 (*i.e.*, bootstrapped).

2 Highlight of Features

AToMPM is a modern, versatile and theoretically sound multi-paradigm modeling environment. It is a tool for modeling any and every part of a system at the most appropriate

Questions