Automatic Test Model Generation for Model Transformations Using Mutation Analysis

A Model-Driven Approach

Author:
Thomas Degueule[1]

Supervisors:
Jean-Marie Mottu[1]
Gerson Sunyé[2]

[1] AeLoS – LINA
[2] AtlanMod – EMN

July 4, 2013

Table of Contents

Introduction

- Model transformations are critical elements of MDE
- Traditional testing techniques need to be adapted to their specificities
- Software testing is an expensive and mainly manual task
- How to help model transformations testers?
    - Generate test models automatically

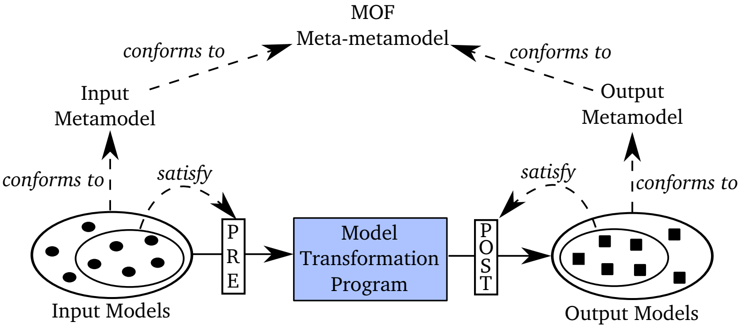
Model-Driven Engineering

Principle

Produce software automatically from high-level models

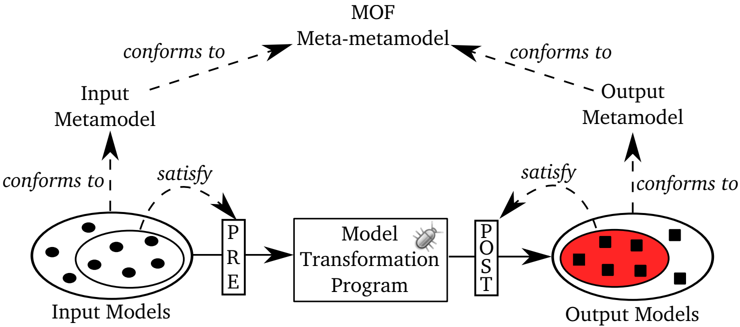
- Each model represents an aspect of the system
- Each model is written in a domain-specific language
- Composition of models forms the whole system
- Models are refined into concrete artifacts
  - Code
  - Tests
  - Documentation
  - Configuration files

Model Transformations



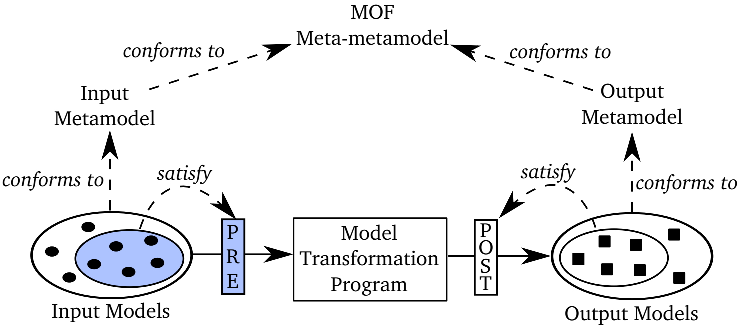
- Written using a *model transformation language* (ATL, Kermeta, . . .)
- Divided into several *transformation rules*
- Usage:
    - Refine abstract models into concrete models
    - Apply design patterns
    - Refactoring. . .

Model Transformations



- Incorrect model transformations lead to corrupted models
- They are used many times in a MDE process
- They are black-box for the end users
- => They need to be trustworthy and thoroughly tested

Model Transformations



- Test data are models: complex and large graph of objects
- They must satisfy many constraints
    - Metamodel conformance
    - Metamodel invariants
    - Transformation preconditions
    - Test intent

Mutation Analysis (1)

Definition

Mutation analysis is a fault-based testing technique used to qualify the test set of a program under test (PUT).

- Faulty versions of the PUT (*mutants*) are created by systematically injecting *one single fault per version*
- These faults are injected using *mutation operators*
- They represent *real faults* a developer may commit

Mutation Analysis (1)

Definition

Mutation analysis is a fault-based testing technique used to qualify the test set of a program under test (PUT).

- Faulty versions of the PUT (*mutants*) are created by systematically injecting *one single fault per version*
- These faults are injected using *mutation operators*
- They represent *real faults* a developer may commit

PUT	Mutants
a = b + c	

Table : The *Arithmetic Operator Replacement* (AOR) operator

Mutation Analysis (1)

Definition

Mutation analysis is a fault-based testing technique used to qualify the test set of a program under test (PUT).

- Faulty versions of the PUT (*mutants*) are created by systematically injecting *one single fault per version*
- These faults are injected using *mutation operators*
- They represent *real faults* a developer may commit

PUT	Mutants
a = b + c	a = b - c
	a = b * c
	a = b / c
	a = b % c

Table : The *Arithmetic Operator Replacement* (AOR) operator

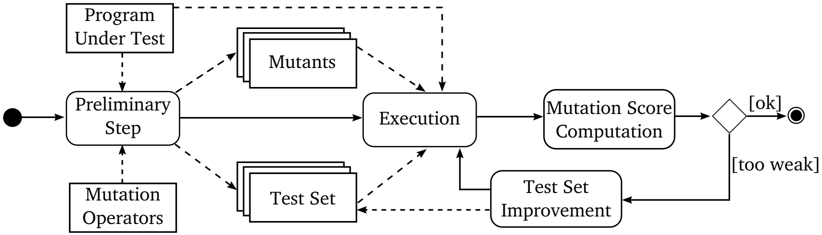
Mutation Analysis (2)

- Mutation analysis supposes the existence of a test set
- Is a test data able to detect the voluntary injected fault?
  - Compare the outputs!
- Let P be the PUT, M one of its mutant and T its test set:
  - If $\exists t \in T : M(t) \neq P(t)$ then the mutant M is *killed*
  - If $\forall t \in T : M(t) = P(t)$ then the mutant M is *alive*

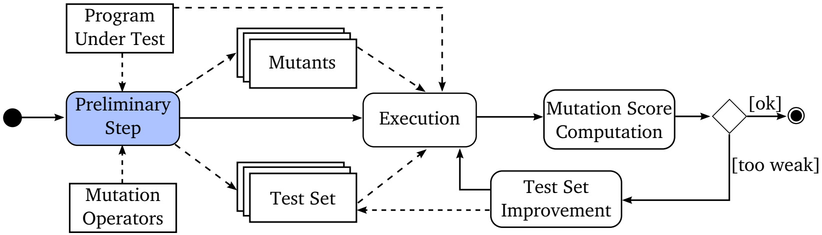
Mutation Score Computation

$$M_{Score}(T) = 100\ x\ \frac{Killed\ \ Mutants}{Total\ \ Mutants - Equivalent\ \ Mutants}$$

Mutation Analysis Process



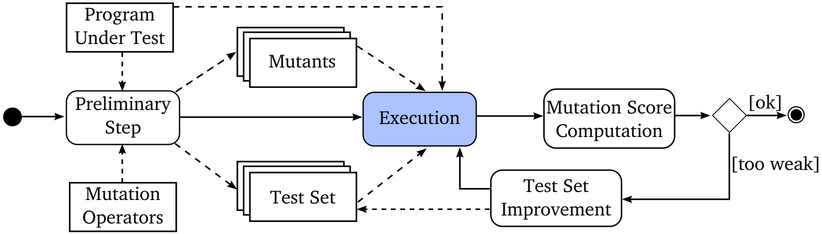
Mutation Analysis Process



Preliminary Step

- Produce the set of mutants
- Based on the language-specific mutation operators of the PUT
- Initial test set provided by the tester or automatically generated

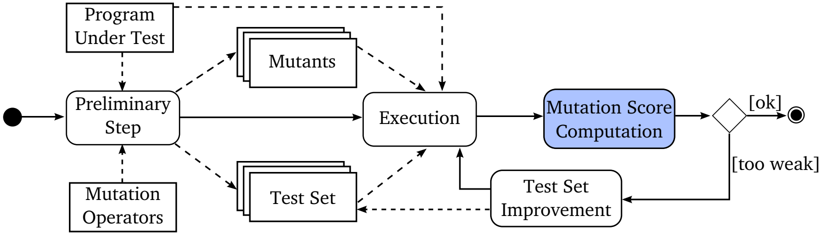
Mutation Analysis Process



Execution

- Compile all the mutants
- Execute all (*test model*, *mutant*) pairs
- Collect the outputs and compare them
- Determine the status of mutants (*killed* or *alive*)

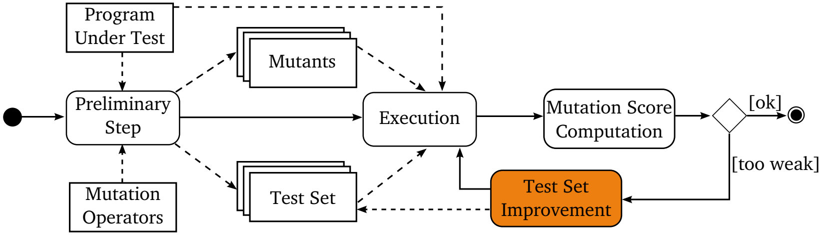
Mutation Analysis Process



Mutation Score Computation

- A human-made test set obtains around 60–75% mutation score
- It is often difficult to reach a 95% mutation score
- Tester must define a threshold beyond which the test set is considered sufficiently efficient

Mutation Analysis Process



Test Set Improvement

- Fully manual task
- Tester needs to determine *why* a mutant has not been killed and *how* to kill it
- Tester needs to analyze test models and create new ones

Mutation Operators for Model Transformations

- Specific mutation operators need to be defined for model transformations
  - *Mutation Analysis Testing for Model Transformations*. Mottu JM., Baudry B. and Le Traon Y. in *Proceedings of the European Conference on Model Driven Architecture* (ECMDA 06)

Category	Description	#
Navigation	Alter the operations of navigation in the models	4
Filtering	Alter the operations of filtering of collection	3
Creation Modification	Alter the creation or modification of elements	3
		10

Table : Mutation Operators for Model Transformations

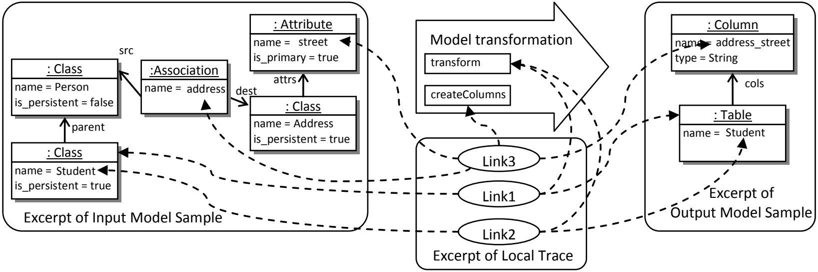
Problematic

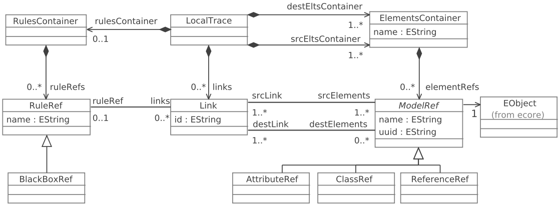
- Building test models from scratch is complex
- Can we *reuse* existing models to create new ones?
- We need to identify relevant test models, and develop heuristics to create new ones

Test Model Improvement Process

1. Which models and which parts of these models are the most relevant?
2. What should the output model look like in order to kill the mutant?
3. How to modify the input model in order to produce this difference?

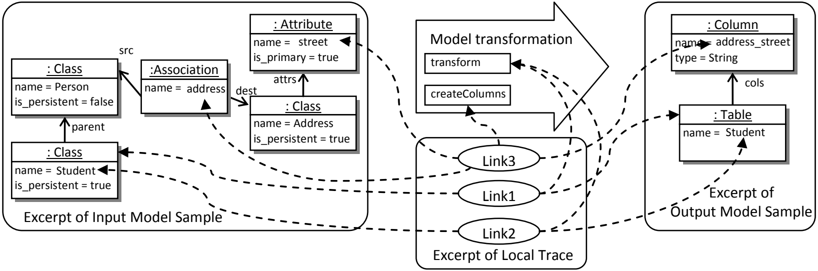
Using Traceability to Collect Information

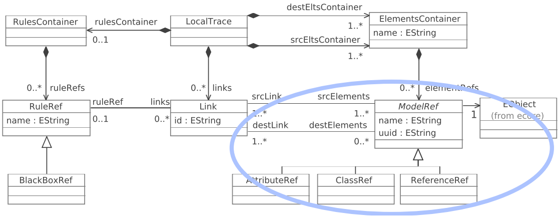




Traceability for Mutation Analysis in Model Transformations. Aranega V., Mottu JM., Etien A. and Dekeyser JL. in Chapters of *Models in Software Engineering*, 2011

Using Traceability to Collect Information





Traceability for Mutation Analysis in Model Transformations. Aranega V., Mottu JM., Etien A. and Dekeyser JL. in Chapters of *Models in Software Engineering*, 2011

Mutation Matrix

- Results of the mutation process are gathered in a mutation matrix

$$
\begin{array}{ccc}
 & & \text{Mutants} \\
 & & \begin{array}{cccc} T_0 & T_1 & \cdots & T_n \end{array} \\
\text{Models} & \begin{array}{c} m_0 \\ m_1 \\ \vdots \\ m_m \end{array} & \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0n} \\ c_{10} & c_{11} & \cdots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m0} & c_{m1} & \cdots & c_{mn} \end{pmatrix}
\end{array}
$$

Mutation Matrix

- Results of the mutation process are gathered in a mutation matrix
- Local trace models are associated to each (*test model*, *mutant*) pair

Mutants

Models

$$\begin{array}{cc} & \begin{array}{cccc} T_0 & T_1 & \cdots & T_n \end{array} \\ \begin{array}{c} m_0 \\ m_1 \\ \vdots \\ m_m \end{array} & \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0n} \\ c_{10} & c_{11} & \cdots & c_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m0} & c_{m1} & \cdots & c_{mn} \end{pmatrix} \end{array}$$

For each (*test model*, *mutant*), we collect:

- The local trace model
- The status of the mutant

Modeling Mutation Operators

- To find out why a mutant remains alive, we need to exploit its semantic difference with the original transformation
- Thus, we need a precise modeling of the mutation operators
- Implementation independent / metamodel independent approach
- Models describe effects upon manipulated data (models)

Modeling Mutation Operators: RSCC Example (1)

> *"The RSCC operator replaces the navigation of one reference towards a class with the navigation of another reference to the same class."*
>
> *Mutation Analysis Testing for Model Transformations*, Mottu *et al.*

```
operation my_rule(assoc : Association, cls : Class) is
do
  assoc.dest := cls

end
```

Figure : RSCC Operator Instanciation Example on a Transformation

Modeling Mutation Operators: RSCC Example (1)

"*The RSCC operator replaces the navigation of one reference towards a class with the navigation of another reference to the same class.*"

Mutation Analysis Testing for Model Transformations, Mottu *et al.*

```
operation my_rule(assoc : Association, cls : Class) is
do
   //assoc.dest := cls
   assoc.src := cls
end
```

Figure : RSCC Operator Instanciation Example on a Transformation

Modeling Mutation Operators: RSCC Example (2)

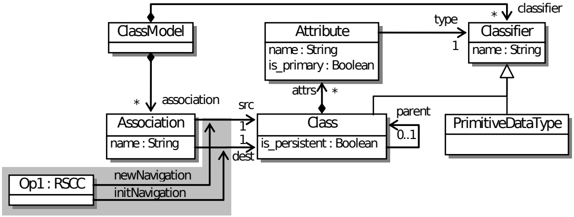


Figure : RSCC Operator Instanciation Example on a Class Diagram Metamodel

Modeling Mutation Operators: RSCC Example (2)

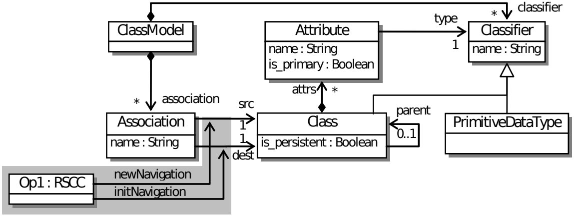


Figure : RSCC Operator Instanciation Example on a Class Diagram Metamodel

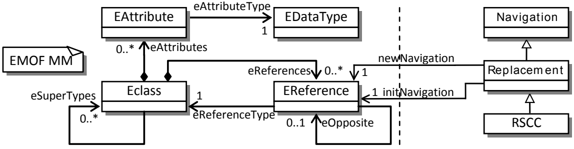
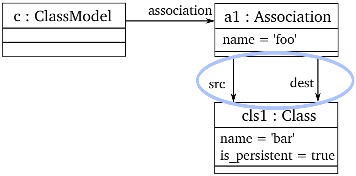


Figure : RSCC Operator Metamodel

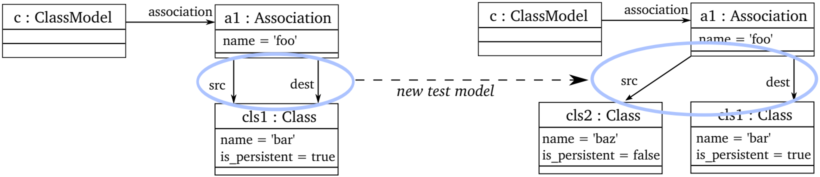
Patterns and Recommendations

- Thanks to the collected informations (trace, mutation models):
  - We can identify specific configurations in the input models that leave the mutant alive
  - We associate recommendations to these patterns that should kill the mutant



Patterns and Recommendations

- Thanks to the collected informations (trace, mutation models):
  - We can identify specific configurations in the input models that leave the mutant alive
  - We associate recommendations to these patterns that should kill the mutant



Experiment : the fsm2ffsm Transformation

- Finite state machine flattening
- Initial test set (9 models) generated with input metamodel coverage techniques
- 148 mutation models $\rightarrow$ 126 mutants

Results & Analysis

- Mutation score from 45% to 100% in 8 iterations
- Gain in terms of elements to be covered: 87%
- 5 mutants killed by automatic application of recommendations
- For 2 mutants, trace models indicated that the mutated rule were not executed
- Only 1 mutant required deeper analysis

Development

- Generic experimentation platform for mutation analysis of model transformations
- Traceability mechanism for Kermeta
- Generation of mutation models based on transformation's metamodels
- Ongoing: Mutant killing constraints to Alloy transformation

Ongoing Work (1)

- Constraint-based generation of test models

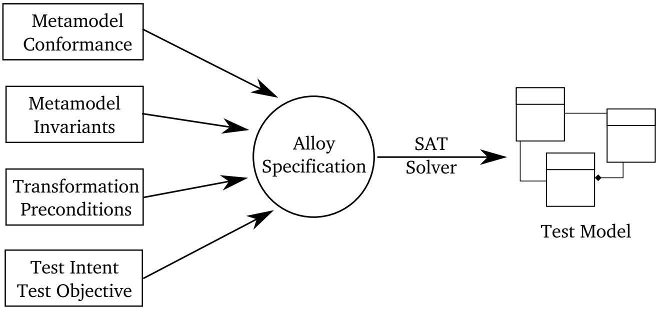


Figure : Constraint-Based Generation of Test Models using ALLOY

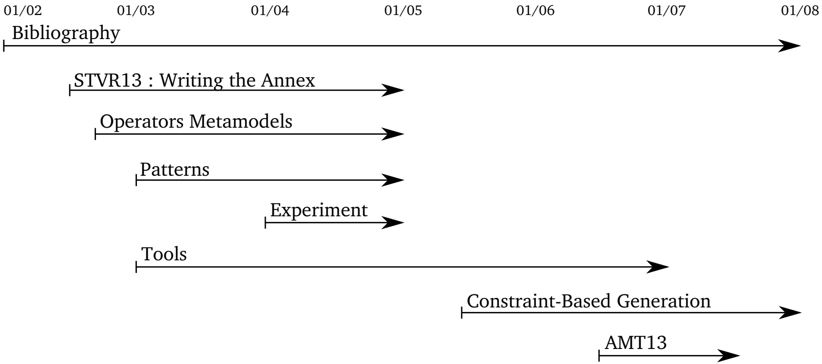
Ongoing Work (2)

- Collaboration with Olivier Finot, PhD student
- Reusing of the experimentation platform in order to study and compare testing oracles
- *Qualifying Oracles in Model Transformation Testing*, in process of writing for the *2nd Workshop on the Analysis of Model Transformations*, MODELS2013

Conclusion

- Ease the tester's work:
  - Trace mechanism drastically reduces the elements to be covered
  - Test models are semi-automatically generated
- MDE approach:
  - Modeling of the mutation operators
  - Results of the process are gathered in a mutation matrix model
- Drawbacks:
  - Trace mechanism must be adapted to each transformation language
  - An initial test set is required for improvement
- Towards a constraint-based generation of test models

The work so far



Towards an Automation of the Mutation Analysis Dedicated to Model Transformation. Aranega V., Mottu JM., Etien A., Degueule T., Baudry B. and Dekeyser JL. submitted to *Software Testing, Verification and Reliability*, 2013