Automatic Test Model Generation for Model Transformations Using Mutation Analysis
A Model-Driven Approach

Author: Thomas Degueule

Supervisors: Jean-Marie Mottu
Gerson Sunyé

1 AeLoS – LINA
2 AtlanMod – EMN

July 4, 2013
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
**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

<table>
<thead>
<tr>
<th>PUT</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b + c</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>PUT</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b + c</td>
<td>a = b - c</td>
</tr>
<tr>
<td>a = b * c</td>
<td>a = b / c</td>
</tr>
<tr>
<td>a = b % c</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigation</td>
<td>Alter the operations of navigation in the models</td>
<td>4</td>
</tr>
<tr>
<td>Filtering</td>
<td>Alter the operations of filtering of collection</td>
<td>3</td>
</tr>
<tr>
<td>Creation Modification</td>
<td>Alter the creation or modification of elements</td>
<td>3</td>
</tr>
</tbody>
</table>

**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

Excerpt of Input Model Sample

Excerpt of Output Model Sample

Excerpt of Local Trace

Using Traceability to Collect Information

Excerpt of Input Model Sample:
- Class name = Person
  is_persistent = false
- Class name = Address
  is_persistent = true

Excerpt of Output Model Sample:
- Model transformation
  createColumns

Excerpt of Local Trace:
- Link1
- Link2
- Link3

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

Mutant Matrix

```
<table>
<thead>
<tr>
<th></th>
<th>T_0</th>
<th>T_1</th>
<th>...</th>
<th>T_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_0</td>
<td>c_{00}</td>
<td>c_{01}</td>
<td>...</td>
<td>c_{0n}</td>
</tr>
<tr>
<td>m_1</td>
<td>c_{10}</td>
<td>c_{11}</td>
<td>...</td>
<td>c_{1n}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>m_m</td>
<td>c_{m0}</td>
<td>c_{m1}</td>
<td>...</td>
<td>c_{mn}</td>
</tr>
</tbody>
</table>
```
Mutation Matrix

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

<table>
<thead>
<tr>
<th>Models</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_0</td>
<td>T_0</td>
</tr>
<tr>
<td>m_1</td>
<td>T_1</td>
</tr>
<tr>
<td>...</td>
<td>T_n</td>
</tr>
<tr>
<td>m_m</td>
<td></td>
</tr>
</tbody>
</table>

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
"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)

![Diagram showing modeling of RSCC operators](image)

**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

```
c : ClassModel
  a1 : Association
    name = 'foo'
    association
  cls1 : Class
    name = 'bar'
is_persistent = true
    src dest```

![Diagram of class model and association](image-url)
• 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 → 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