Towards Systematic Mutations for and with ATL Model Transformations*

Patrick Sommer[1] and Carola Gabriel[2] and Martin Keiblinger[3]

[1] Mautner Markhof-Gasse 58/4/31, 1110 Wien
e0925011@student.tuwien.ac.at
MatrNr.: 0925011
[2] Mustergasse 54/4/3, 1030 Wien
matthias@tuwien.ac.at
MatrNr.: 0426553
[3] Mustergasse 54/4/3, 1030 Wien
matthias@tuwien.ac.at
MatrNr.: 0426553

Abstract. This abstract summarizes the content of this paper in about 70 to 150 words. ...

* This work has been created in the context of the course "188.952 Advanced Model Engineering (VU 2,0)" in SS15.

Table of Contents

1 Introduction

The broader goal of this work is to create an introduction to a specific field of software testing. Software testing is a process, or a series processes engineered to check if a program does what it is designed to do and that id does not do anything unintended. [**?**] Model based testing (MBT) is a variant of testing. Test cases are not written by the programmer directly. The programmer creates a model of the requirements and in a second step the test cases are generated on base of the model. [**?**]

Mutation testing is a fault-based testing technique. It applys changes to the input and creates a mutant. A mutant represents a faulty program. In the best case these changes, which are applied by the mutator, represent mistakes a programmer would make. [**?**]

The basic idea is of mutation testing is not to test the resulting software itself but the test cases. Good test cases should be able to identify mutants. Identifying means recognizing differing results of the original system under test (SUT) or mutants. [**?**]

The process of mutation testing consists of these components:

- *Test data* as input for the original programm P and its mutants.
- The original program P
- The *mutants* of P.
- An *oracle* which is able to decides if results differ and which is therefore able to identify mutants.

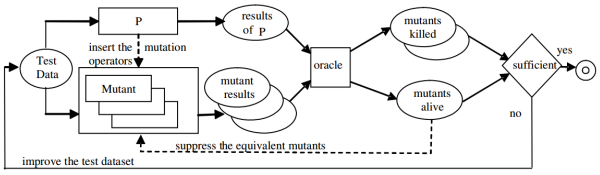

Fig. 1. The mutation testing workflow contains a feedback loop. [**?**]

The goal of the process1 is to *kill* or identify faulty versions of P. If a mutant outputs the same data as the P for the same input data it's called *equivalent*. In this case this mutant has to removed from the set of mutants under test.

The last step is to assess how good the tests are and check if they should be improved. Assume KM as the set of the killed mutants, M is the set of all mutants and EM is the set of all identified, equivalent mutants. Then the mutation score MS is calculated like this: [**?**]

$$MS = \frac{|KM|}{|M| - |EM|} \tag{1}$$

If this value is to small the tests have to be improved.

The success of this method depends on the set of mutants used in the process. Manual creation of mutants is a tedious and time consuming task. Therefore a quick, reliable and efficent creation of mutants is proposed in [**?**].

Troya et. al. build upon ATL and higher order transformations (HOT) to create transformations to automatically generate mutants.

This report show what additional transformation have been developed and what their goal is. The scope of this work is only on the mutation generation in the whole process.

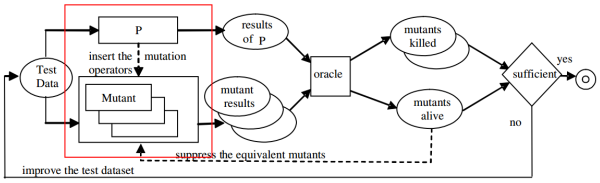

Fig. 2. The mutation testing workflow contains a feedback loop. [**?**]

2 General

Model transformations play an important role in the Model Driven Engineering (MDE) approach. Developing model transformation definitions is expected to become a common task in model driven software development. [**?**] In this part of the paper we want to explain the basics of the requirements we needed for Mutations for and with ATL Model Transformations.

2.1 Model transformation in MDE

Model transformation is an important technique in software development, espacially in Model-Driven Software Development (MDSD) and Model-Driven Software Development (MDA). There exists different types of model transformations like Model-To-Model Transformation and Model-To-Text Transformation.

2.2 ATL

ATL is a model transformation language containing a mixture of declarative and imperative constructs. ATL is applied in the context of the transformation pattern shown in . In this pattern a source model Ma is transformed into a target model Mb according to a transformation definition mma2mmb.atl written in the ATL language. The transformation definition is a model conforming to the ATL

metamodel. All metamodels conform to the MOF. ATL is a hybrid transformation language. It contains a mixture of declarative and imperative constructs. We encourage a declarative style of specifying transformations. The declarative style of transformation specification has a number of advantages. It is usually based on specifying relations between source and target patterns and thus tends to be closer to the way the developers intuitively perceive a transformation. This style stresses on encoding these relations and hides the details related to selection of source elements, rule triggering and ordering, dealing with traceability, etc. Therefore, it can hide complex transformation algorithms behind a simple syntax. [?]

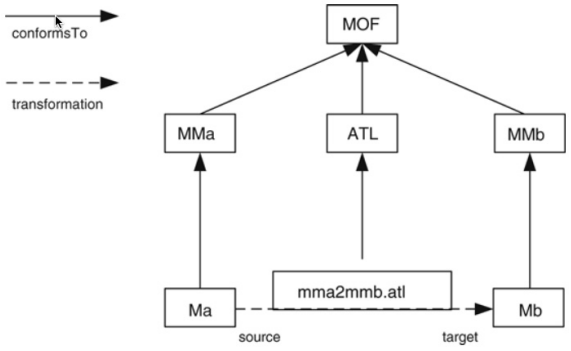

Fig. 3. Overview of the ATL transformational approach

[?]
[?]
[?]

In the following you can see a short example of a ATL transformation:

```
//Start Program
module Entities2Forms;
create OUT : Forms from IN : Forms;

rule EntityModel2FormModel {
from
em : Forms!EntityModel
to
fm : Forms!FormModel (
)
}
//End Program
```

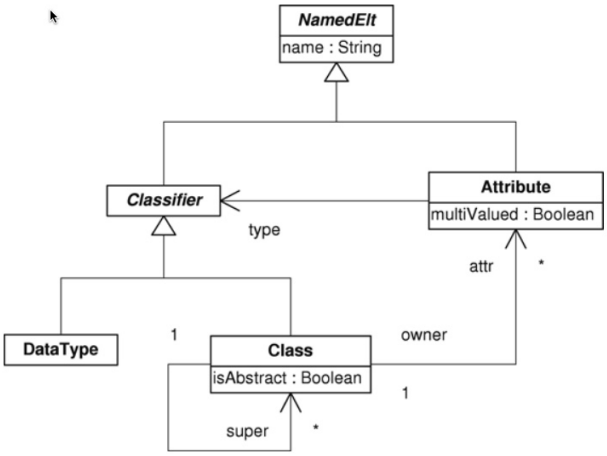

Fig. 4. Class metamodel

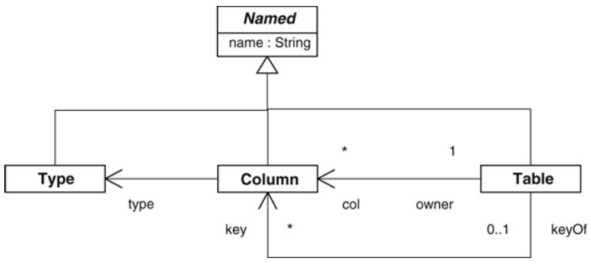

Fig. 5. Relational metamodel

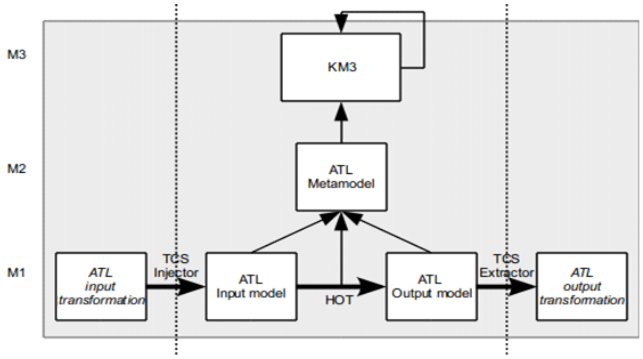

Fig. 6. Sample schema of a HOT for transformation modification in ATL

This ATL file shows a tranformation Entities2Form. The source model is the 'IN' model and the target model is 'OUT'. The rule mapped the elements EntityModel to FormModel.

2.3 High Order Transformations

Higher order transformations is a model transformation such that its input and/or output models are themselves transformation models. [**?**]

Therefore a transformation model is:

- The input of a HOT
- The output of a HOT
- Or it is both

Tisi et. al identified four transformation patterns for HOTs:

- **Transformation Synthesis** These HOTs generate transformations. If any input is used it's not a transformation.
- **Transformation Analysis** take transformations as input and generate data of different kinds as output. The output is never a transformation.
- **Transformation (De)composition** is the integration of multiple transformations of the input and/or integration of transformations as output.
- **Transformation Modification** is defined by modifing an input transformation and generating an output transformation.

The focus of this work is on *Transformation Modifications* (short TM).

2.4 Mutations in Model Transformations

[**?**]

3 Implementation

3.1 Transformations

Binding Addition
Implementation
Discussion
Deletion
Implementation
Discussion
Change
Implementation
There are two types of major types of binding values in the ATL metamodel:

- Primitive Expressions
- Variable Expressions

The primitive expressions are directly assignable in the ATL rules. Variable expressions have to be composed from the available objects in the metamodel. (See 7)

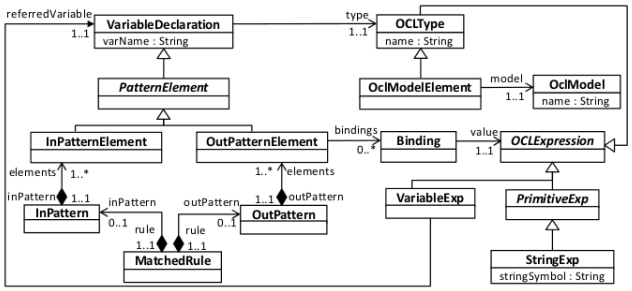

Fig. 7. Excerpt of the ATL metamodel

The first example shows how to change the value of all String Expressions. The output value will be the String "Hello".

To achieve this, we first have to search for all bindings with the value type "ATL!StringExp" and set this value to our defined string value.

```
rule ValueChangeBinding_All{
        from
         b : ATL!Binding (b.value.oclIsTypeOf(ATL!StringExp))
        to
         c : ATL!Binding (
```

```
                value <- newStringExp
        ),
        newStringExp : ATL!StringExp(
                stringSymbol <- 'hello'
        )
}
```

As an extension to this approach, in a second step we want to overwrite all String expressions to the first String value of the input model. This only make sense, if we have more than one string expression, so we added a condition to do this.

```
rule ValueChangeBinding_AllSameStringValue{
        from
         b : ATL!Binding(
                b.value.oclIsTypeOf(ATL!StringExp) and
                ATL!StringExp.allInstances() -> size() > 1
         )
        to
         c : ATL!Binding(
                value <- stringexpression
         ),
         stringexpression : ATL!StringExp(
                stringSymbol <- (ATL!StringExp.allInstances()
                        -> first()).stringSymbol
        )
```

The third approach in this category was to switch the first with the last value binding. For this we fist filter only the OutPatternElements which have more than one binding and have only StringExpression bindings.

As a second step we exclude the first and the last binding from the binding-list to replace them with our new bindings with the prepend and append commands. As a third step we have to create our new binding-elements.

Such a binding element needs three values: the corresponding OutPatternElement, the property name and the value.

As outPatternElement and propertyName we take the original outPatternElement and propertyName, as value, we take value from the last binding for our new first elemen and the value from the fist binding for our new last element.

Important to achieve correct values is, that we use the original binding list from the "from"-clause of the rule.

```
rule ValueChangeBinding_Switch{
from
ope : ATL!OutPatternElement (
        ope.bindings
                -> forAll(e | e.value.oclIsTypeOf(ATL!StringExp)) and
                ope.bindings -> size() > 1
)
```

```
to
ope2 : ATL!OutPatternElement (
        bindings <- ope.bindings -> excluding (ope.bindings -> first())
                -> excluding (ope.bindings -> last())
                -> prepend (bindingNewFirst)
                -> append (bindingNewLast)
),
bindingNewLast : ATL!Binding (
        outPatternElement <- ope2,
        propertyName <- (ope.bindings -> last()).propertyName,
        value <- (ope.bindings -> first()).value
),
bindingNewFirst : ATL!Binding (
        outPatternElement <- ope2,
        propertyName <- (ope.bindings -> first()).propertyName,
        value <- (ope.bindings -> last()).value
)

}
```

Discussion

Out Pattern Element Addition

Implementation
Discussion
Class change
Implementation
Discussion

In Pattern Element Deletion

To delete an InPatternElement it is not enough to delete only the InPatternElement itself. You have to deal at least with one important effect. The usage of this InPatternElement in the Bindings of the OutPatternElements. For better understanding this problem, I will give a short example.

```
rule PNMLDocument {
        from
                e : PetriNet!PetriNet,
                f : PetriNet!PetriNet
        to
                n : PNML!PNMLDocument
                (
                        location <- e.location,
                        xmlns <- uri,
                        nets <- net
                )
```

```
rule PNMLDocument {
        from
                f : PetriNet!PetriNet
        to
                n : PNML!PNMLDocument
                (
                        location <- .location,
                        xmlns <- uri,
                        nets <- net
                )
```

If you delete the IPE with the variableName e, the binding location ¡- e.location will have an unreferredVariable in its value part. So the generated model is not valid any more. To avoid this, you have to delete all bindings with unreferred variables in a second step of your high order transformations.

The implementation of the deletion of an IPE looks like this:

```
rule DelteIPE {
  from ipe : ATL!InPatternElement (
    ipe.inPattern.elements -> size() > 1 and
    ipe = ipe.inPattern.elements -> last()
  )
```

The following deletion of Binding is realized with this code:

```
rule DelteNavigationBinding{
from b : ATL!Binding(
        b.value.oclIsTypeOf(ATL!NavigationOrAttributeCallExp) and
        b.value.source.referredVariable.oclIsUndefined()
)
to
}
rule DeleteCollectionBinding{
from b:ATL!Binding(
        b.value.oclIsTypeOf(ATL!CollectionOperationCallExp) and
        b.value.source.oclIsTypeOf(ATL!NavigationOrAttributeCallExp) and
        b.value.source.source.referredVariable.oclIsUndefined()
)
to
}
```

First of all you need to know, that there are several types of possible binding values - the CollectionOperationCallExp and the NavigationOrAttributeCall-Exp. As they have a different strucutre and to deal with both of them. you have to write two rules to delete the corresponding bindings.

Implementation

Discussion

Class change

Implementation
Discussion
Explain what we did and why.

3.2 Related transformation

Explain what other transformations would make sense.

4 Conclusion

5 Bibliographic Issues

5.1 Literature Search

Information on online libraries and literature search, e.g., interesting magazines, journals, conferences, and organizations may be found at `http://www.big.tuwien.ac.at/teaching/info.html`.

5.2 BibTeX

BibTeX should be used for referencing.

The LaTeX source document of this pdf document provides you with different samples for references to journals [**?**], conference papers [**?**], books [**?**], book chapters [**?**], electronic standards [**?**], dissertations [**?**], masters' theses [**?**], and web sites [**?**]. The respective BibTeX entries may be found in the file `references.bib`. For administration of the BibTeX references we recommend `http://www.citeulike.org` or JabRef for offline administration, respectively.