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] Leberweg 14/3/16, 1110 Wien
e0825992@student.tuwien.ac.at
MatrNr.: 0825992
[3] Mustergasse 54/4/3, 1030 Wien
matthias@tuwien.ac.at
MatrNr.: 0426553

Abstract. Faults in model transformations will produce faults in the generated models. The correction of the defective models is often very expensive. This result affects the quality of the end product. This is why model transformations have to be correctly tested to maintain product quality. For this, mutation testing is a popular technique. The mutation testing of model transformations has to be developed and for this a suite of mutation operators for the Atlas Transformation language (ATL) is needed. In this paper a set of mutation operators are implemented and the solutions of the implementations are discussed and evaluated regarding effectiveness.

* 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

Model transformation allows to synthesize software artifacts from model definitions and ease other software engineering tasks by automating them.

This abstraction eases incremental processes like [4]:

- Reverse engineering models e.g. in the process of replacing a legacy system. The resulting artifact can be tested and if results of the legacy and the new system are found only the model has to be changed.
- Refactoring models

Therefore the quality of the overall software solution is determined by the quality of the model transformations and the resulting models.[5] As a consequence testing the transformations for correctness is an essential part to achieve the best possible quality of the software. [11]

To check the quality of a software it has to be tested. Software testing is a process, or a series of engineered processes to check if a program does what it is designed for and that it does not do anything unintended. [7] Model based testing (MBT) is a variant of this software testing process. Test cases are not written by the programmer directly. The programmer only creates a model of the requirements and in a second step the test cases are generated based on this model. [12]

Mutation testing or mutation analysis is a fault-based testing technique. It applies changes to the input model and creates a mutant. Every mutant represents a faulty program. In the best case these changes, which are applied by the mutator, represent mistakes a programmer would make. It has been proven that mutation testing is useful as testing approach but also as: [4]

1. Generate input models as test data.
2. Generate mutants of model transformations.

1.1 Problem

To make mutation testing an effective testing method a complete set of mutation operators and a large number of mutated model transformations have to exist. Due to the needed size of these sets it is too expensive to create them manually. Also the execution of the different models against the test data is very time-consuming.

Therefore automation is required for:

1. generation of a set of mutation operators.
2. generation of mutated model transformations.

Furthermore, the computational costs of the executions have to be lowered.

[4] Sendall:2003
[5] Hutchinson:2011

1.2 Contribution

Troya et. al. [11] contributed three aspects.

1. They have defined a general language-centric synthesis approach by defining a set of mutation operators based on ATL.
2. They made a concept and built a first version of a framework utilizing higher-order transformations for generating mutants. [10]
3. They integrated techniques [1] for incremental model transformation execution in their framework.

We contribute further implementations of mutation operators and small tests.

2 Background

2.1 Model transformation in MDE

This work is focused on mutation testing for model transformations. The basic idea of mutation testing in software engineering is not to test the resulting software itself but the test cases.

Good test cases should be able to identify mutants. In the jargon of mutation testing the mutations are *killed*. Killing means recognizing differing results of the original system under test (SUT) and tested mutants. [6]

The process of mutation testing consists of four 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 decide if results differ and which is therefore able to identify mutants.

The goal of the process, shown in Figure 1, is to *kill* or identify faulty versions of the original program. If a mutant outputs the same data for the same input data as the original program it is called *equivalent*. If the results are the same, then the input data has to be improved. If the results from the mutant and the original program differ, then the mutant is killed.

The last step is to assess how good the tests are and check if they should be improved. Assume KM is 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: [4]

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

If this value is too 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 [11].

Troya et al. [11] build upon ATL and higher order transformations (HOT) to create transformations to automatically generate mutants.

This report shows which additional transformations have been developed and what their goals are. The scope of this work is the part of the figure 1 that is marked, which deals with the generation of mutants for model transformations.

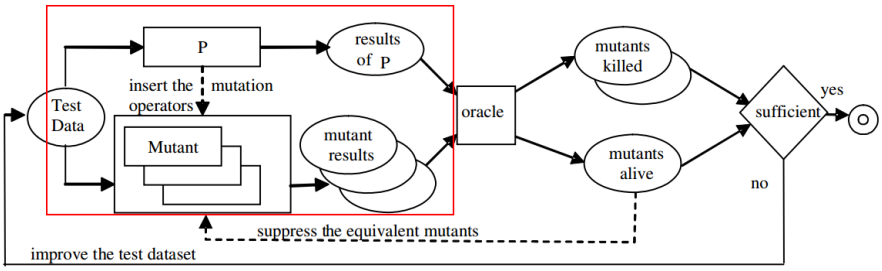

Fig. 1. The mutation testing workflow contains a feedback loop. The red marked area is the part this paper contributes to. [6]

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. [3] 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.

2.2 Mutation Testing

Speaking in broader terms, model transformation is the process of synthesizing one or more models from one or more input models. To successfully generate output-models it needs a clear understanding of the syntax and the semantic of these models and the relationship between their different entities. Model driven engineering provides a formal framework to define this prerequirements. [8]

At the highest level two main categories of model transformations exist: [2]

1. Model to model transformations
2. Model to code transformations

Model to code transformations take models as input and generate source code as output. There exists two different approaches to synthesize the generated source code. On the one side there is the visitor based approach, which is driven by a model traverse engine which outputs code artifacts when it visits specific points in the model hierarchy. The other approach is the template based

approach. Therefore a template consists of constant text and parts which dynamically generate text from the input model. The dynamically part expands iterativly. [2]

For model to model transformations there exist multiple strategies:

- The *Direct-manipulation approach* offers an internal model representation and tools to manipulate it.
- The *relational approach* uses relations, in the sense of the mathematical concept, and mapping rules. In combination with a declerative logic for defining constraints, this is used to produce executable transformations.
- The *graph-transformation-based approach* utilizes graph theory. It models entities as nodes and edges in the graph to define model transformations.
- The *structure-driven approach* provides the user with a framework to operate on a hierarchy to copy objects and their attributes.
- The *hybrid approach* is the combination of two or more of the above mentioned concepts.

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 figure 2. 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. This transformation definition is a model conforming to the ATL metamodel. All metamodels conform to the MOF.

ATL is a hybrid transformation language. It is encouraged to use 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. [3]

As an example of an ATL transformation, Listing 1.1 shows an excerpt of the Class2Relational transformation that transforms simple class models to relational models. Input models are defined conforming to the class metamodel (cf. Figure 3), while the generated output models conform to the relational metamodel shown in Figure 4.

Listing 1.1. Simple example for a ATL transformation

```
1 //Start Program
2 module Class2Relational;
3 create OUT : Relational from IN : Class;
4
5 rule Class2Table {
6     from
7         c : Class!Class
```

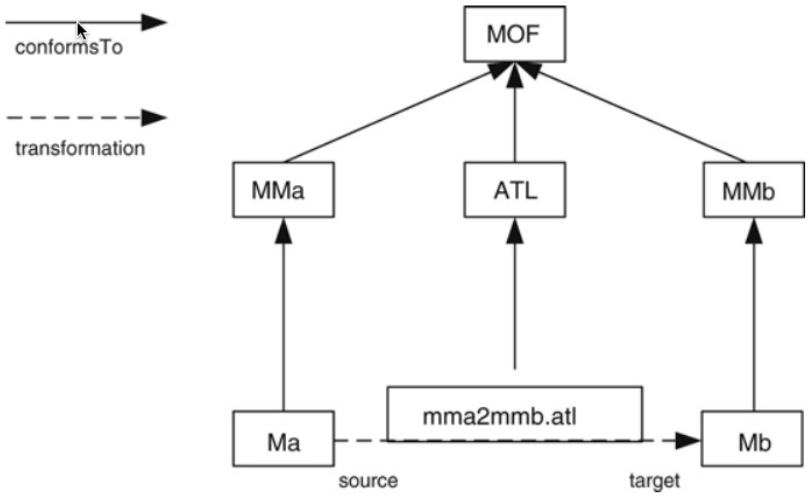

Fig. 2. Overview of the ATL transformational approach (from [3])

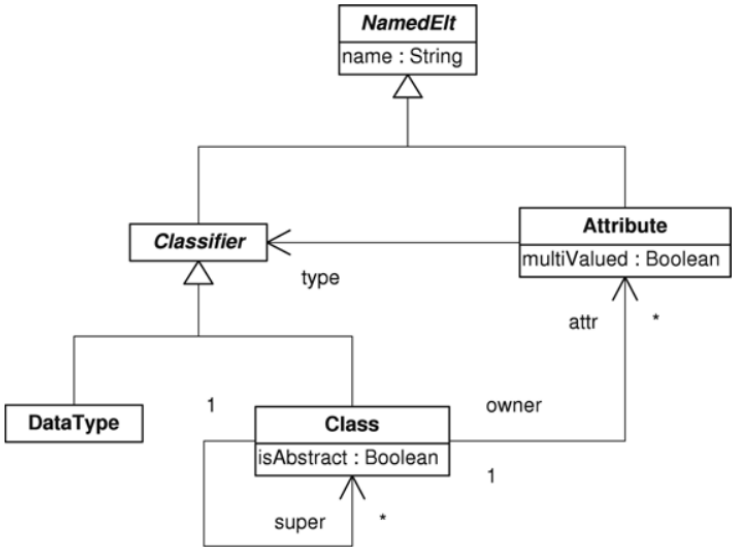

Fig. 3. Class metamodel (from [3])

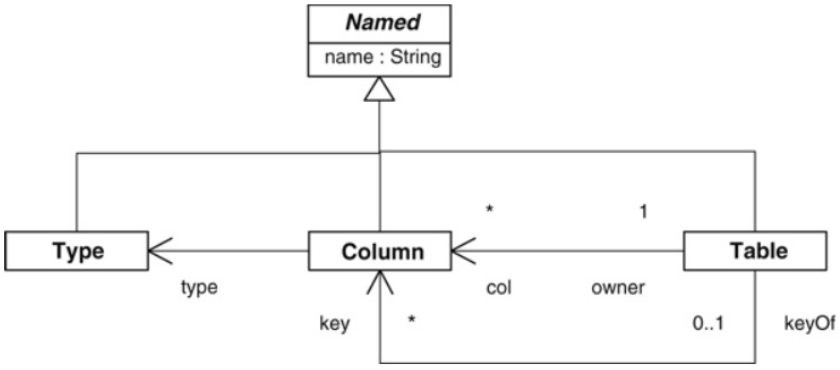

Fig. 4. Relational metamodel (from [3])

```
 8      to
 9          out : Relational!Table (
10              name <- c.name
11          )
12  }
13  //End Program
```

The source model is the 'IN' model and the target model is 'OUT'. The rule actually maps elements of type Class and creates elements of type Table.

Higher Order Transformations Higher-order transformations (HOTs) are defined as model transformations where the input and/or the output models are transformation models themselves. [9]

Figure 5 shows a sample schema of a higher order transformation for transformation modification in ATL where you can see a good overview for the model transformation with transformation models as input and output models.
[5]

Therefore a transformation model can be:

- The input model of a higher order transformation
- The output model of a higher order transformation
- Both, the input and the output model

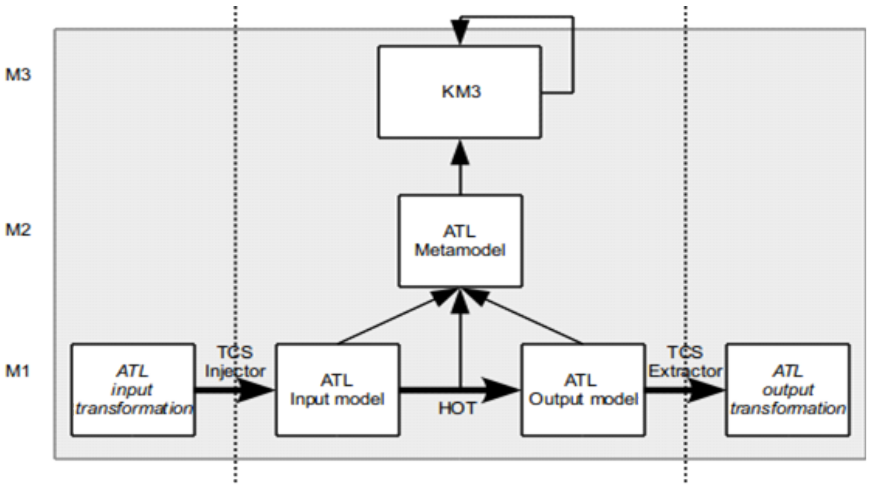

Fig. 5. Sample schema of a Higher Order Transformations for transformation modification in ATL

[5]

Tisi et. al [9] identified four transformation patterns for Higher order transformations:

- **Transformation Synthesis** These Higher order transformations generate transformations. If any input is used it is not a transformation.
- **Transformation Analysis** takes transformations as input and generates data of different types 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).

Another possibility of a Higher order transformation is a *second-order HOT*. That means that a Higher order transformation will also be the input/output of the second-order HOT. Figure 6 exemplifies the use of model transformations, HOTs and second-order HOTs.

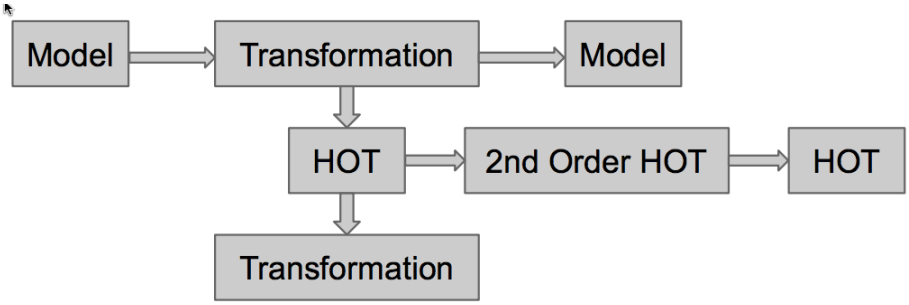

Fig. 6. Sample schema of a Second Higher Order Transformation

2.3 Mutations in Model Transformations

In order to identify the possibilities of mutations in languages we need to know about the concepts of the languages which can be mutated. This mutation possibilities are presented in the referenced paper [11]. For a better overview the following table shows the possible mutations for ATL transformations.

In the column *Consequences* of table 1 there are some notices about the consequences of a mutation. In the referenced paper [11] the consequences are described as follows: When a transformation is mutated, it has an effect on the output model that is generated for the same input model as with the original transformation.

Concept	Mutation Operator	Consequence
Matched Rule	Addition Deletion Name Change	OA;[RA] OD;[RD]
In Pattern Element	Addition Deletion Class Change Name Change	OA;[RA] OD;[RA] OD,OA;[RD];[RA]
Filter	Addition Deletion Condition Change	OD OA OA;OD
Out Pattern Element	Addition Deletion Class Change Name Change	OA;[RA] OD;[RD] OR;[RA];[RD]
Binding	Addition Deletion Value Change Feature Change	OP;[RA] OPN;[RD] OPM;[RA];[RO]

Table 1. Mutations identified for ATL transformations [1]

There can be completely new objects that were not present before (OA: Object Addition), some objects can be deleted (OD: Object Deletion), some can be replaced (OR: Object Replacement) and other objects can be modified. They are labeled as OPI: Object Property Initialized or OPN: Object Property set to Null. That means that a property of an obejct is initialized or set to null. Another possibility is to modify only the value of objects property (OPM: Object Property Modified). For OR: Object Replacement it is important to know that Object Replacemnt can also be seen as the deletion and addition of an object. Relationships between objects can also be added (RA: Relationship Added) or deleted (RD: Relationship Deleted). [11]

For our implemented solutions the consequences OPI: Object Property Initialized, OPN: Object Property set to Null, OPM: Object Property Modified, OA: Object Addition, OR: Object Replacement and OD: Object Deletion are important. In case of relationships we have to know that the modification of an object implies automatically the modifictions of relationships and that is why also RA: Relationship Added or RD: Relationship Deleted are important for our implemented solutions.

3 Mutations implemented

The overall goal of this work is to implement seven transformations. The following table shows the definition of these goals and the current implementation status.

Nr.	Concept	Mutation operator	Status
1	Binding	Value change	Done
2	Binding	Value change	Done
3	Binding	Addition	Done
4	Binding	Deletion	Done
5	Out pattern element	Class change	Done
6	Out pattern element	Addition	Done
7	In pattern element	Deletion	Done
8	In pattern element	Class change	Done

Table 2. The implementation goals

3.1 Implemented binding mutations

To understand how the binding mutations work you need to know, that there are two types of bindings in the ATL metamodel:

- Those which contain only an object of a primitive type
- Those which contain variable expressions

The primitive expressions, like String- or IntegerExpressions, are directly assignable in the ATL rules. Variable expressions have to be composed from the available objects in the metamodel (see figure 7).

Bindings are used for the initialization of either *attributes* or *references*. Attributes are primitive values whereas references are relations to other model elements (variable expressions). Mutation *1* and *2* in table 2 are attribute binding mutations. Mutation *3* is a reference binding mutation.

Deletion Deleting a binding is the implicit omission of an attribute or reference assignment in the *to*-part of the ATL rule. The example in listing 1.2 demonstrates the simple structure of delete bindings.

Listing 1.2. Definition of Delete-Binding.

```
1 rule DeleteBinding{
2   from b: ATL!Binding
3   to
4 }
```

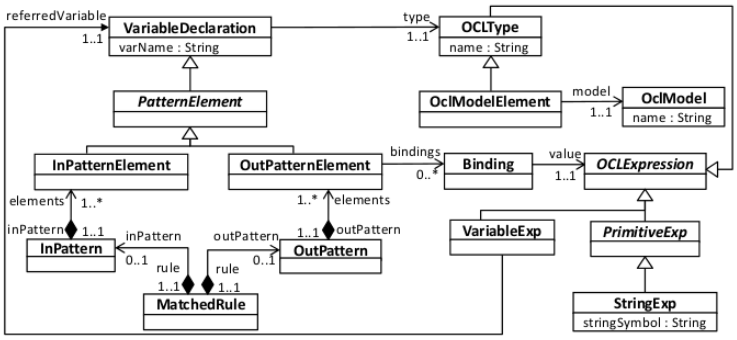

Fig. 7. Excerpt of the ATL metamodel

Addition The addition of a binding is the introduction of a new assignment in the *OutPatternElement*. This does not work for an arbitrary type as there are type constraints of the metamodel of the OutPatternElement.

Therefore the rule has to implement a type lookup to take the set of possible properties of a specific type into account. This is not possible in the HOT because the type information of the specific metamodels are missing.

Troya et al proposed in their paper [1] second order HOTs which are applied to the first order HOTs. The result of the first order HOTs transformations contain type information which is available in the second order HOTs.

This results in a two pass transformation:

1. Add a binding with a dummy name (e.g. NewBindingPropertyName).
2. Lookup available bindings and replace the dummy name with an existing name or type from the corresponding input metamodel.

Additionally the rule has to take care to prevent the same OutPatternElement to be transformed more than once by filtering out entities which already contain the dummy binding.

This proposed algorithm is showed by listing 1.3

Listing 1.3. AddBinding-Definition

```
1 rule AddBinding{
2   from
```

```
3   ope : ATL!OutPatternElement (
4     ope.bindings -> forAll( b | b.propertyName <> 'NewBindingPropertyName'
          )
5   )
6   to
7   ope2 : ATL!OutPatternElement (
8     bindings <- ope.bindings -> append(bindingNewElement)
9   ),
10  bindingNewElement : ATL!Binding (
11    outPatternElement <- ope2,
12    propertyName <- 'NewBindingPropertyName',
13    value <- newValue
14  ),
15  newValue : ATL!StringExp (
16    stringSymbol <- 'testvalue'
17  )
18 }
```

The second order HOT does a lookup for a general supertype in the metamodel. In listing 1.4 the rule chooses an attribute, called *EStructuralFeature*, which is added to all subtypes.

Listing 1.4. AddBindingNames-Definition.

```
1  rule AddBindingNames {
2  from b : ATL!StringExp (
3      b.stringSymbol = 'NewBindingPropertyName'
4  )
5  using{
6    classes : Sequence(OUT_MM!EClass)
7      = OUT_MM!EClass.allInstancesFrom('OUT_MM')
8      -> select(c | c.getESuperTypes() -> size() <= 0);
9  }
10 to b2 : ATL!StringExp (
11   stringSymbol <- classes
12     -> first().getEStructuralFeatures() -> last().name
13 )
14 }
```

Detailed description of addbinding reference: For example there exist two metamodels, where models confirming to the first (left) one are transformed to models confirming to the second (right) one (shown in figure 8).

For the next step we want create a new binding with a reference (for example: b <- c1.a). First we have to search for a reference from the model. For this reference it is important that there is no binding in the original transformation. Furthermore we have to store the attributes propertyName (‘b‘) and EType (‘T2‘). Then we have to search for a rule from the original transformation which has as a first OutPatternElement one with the ‘T2‘ and next we have to store the type of the corresponding InPatternElement which is ’C2’. For the next step we need a reference from the input metamodel which has type ’C2’ and store its name (‘a‘).

For the first implementation of the program it was not implemented to use the original transformation as input of the second order HOT. For this reason we couldn’t search for a rule with the corresponding OPE in the second order HOT, but only in the first order HOT But in the context of the first order HOT there is no access to the class names and attributes of the input and output

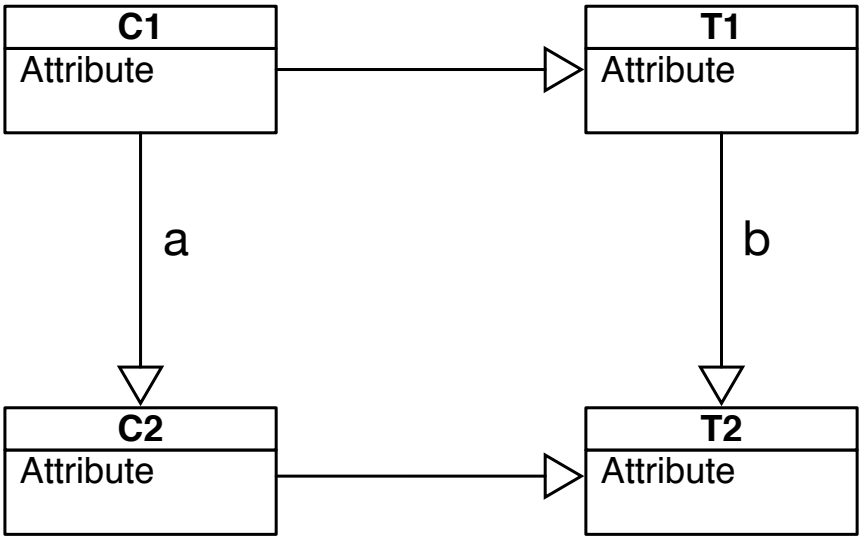

Fig. 8. Example of a simpel metamodel transformation

model (‘T1‘, ‘T2‘, ‘C1‘, ‘C2‘). Since, in order to implement such mutation, the second-order HOT needs information of the ATL transformation to be mutated, we needed to modify and extend the overall approach presented by Troya et al. As shown in figure 9, the second-order HOT has now one more input: the ATL transformation to be mutated. This allows us to define mutations where we need information of rule inter-dependencies. In listing 1.5 you can see an example about how to find out the correct value for our new reference binding.

Listing 1.5. AddBindingValue-Definition.

```
1  rule AddBindingReferenceValue2{
2      from a : ATL!StringExp(
3          a.stringSymbol = 'testname'
4      )
5      using{
6
7          classToAdd : Sequence(OUT_MM!EClass) = OUT_MM!EClass.
               allInstancesFrom('OUT_MM')
8              -> select(c | c.getEStructuralFeatures() -> exists ( f | f.
                   oclIsTypeOf(OUT_MM!EReference))
9                  and c.getEStructuralFeatures() -> exists ( f |
                       allBindingsForClass -> forAll( b | b.propertyName.
                       debug('bindingname') <> f.name.debug('f.name')).debug
                       ('forall'))
10             );
11         allBindingsForClass : Sequence(IN_ATL!Binding) = IN_ATL!Binding.
               allInstancesFrom('IN_ATL').debug('bindings');
12         bindingB : OUT_MM!EStructuralFeature = classToAdd -> first().
               getEStructuralFeatures() -> select (f | allBindingsForClass
               -> forAll(b | b.propertyName <> f.name)) -> first();
13         typeB : OUT_MM!EClass = bindingB.getEType();
```

```
14
15          c : IN_ATL!MatchedRule = IN_ATL!MatchedRule.allInstancesFrom('
                IN_ATL') -> select(r | r.outPattern.elements -> first().type.
                name = typeB.name) -> first().debug();
16          ref : IN_MM!EReference = IN_MM!EReference.allInstancesFrom('IN_MM'
                ) -> select(r | r.eReferenceType.name = c.inPattern.elements
                -> first().type.name) -> first().debug();
17
18
19      }
20      to a2 : ATL!StringExp(
21          stringSymbol <- ref.name
22      )
23  }
```

Value change The simpliest value change mutation is changing a primitive expression like a StringExp. The example in listing 1.6 demonstrates how to change a primitive string expression to a constant ("Hello").

This is achieved by looking up all bindings with the *ATL!StringExp* type. The next step is to set it to the constant.

Listing 1.6. ValueChangeBinding-Definition.

```
1  rule ValueChangeBinding_All{
2      from
3       b : ATL!Binding (b.value.oclIsTypeOf(ATL!StringExp))
4      to
5       c : ATL!Binding (
6          value <- newStringExp
7       ),
8       newStringExp : ATL!StringExp(
9          stringSymbol <- 'hello'
10      )
11 }
```

A simple extension of the constant value change algorithm is to change the constant to a value of the actual model. The example in listing 1.7 changes all string expressions to the first string value of the input model. In case there's no string expression the element is ignored.

Listing 1.7. ValueChangeBinding-Definition using same constant string value.

```
1  rule ValueChangeBinding_AllSameStringValue{
2      from
3       b : ATL!Binding(
4          b.value.oclIsTypeOf(ATL!StringExp) and
5          ATL!StringExp.allInstances() -> size() > 1
6       )
7      to
8       c : ATL!Binding(
9          value <- stringexpression
10      ),
11      stringexpression : ATL!StringExp(
12         stringSymbol <- (ATL!StringExp.allInstances()
13             -> first()).stringSymbol
14     )
```

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

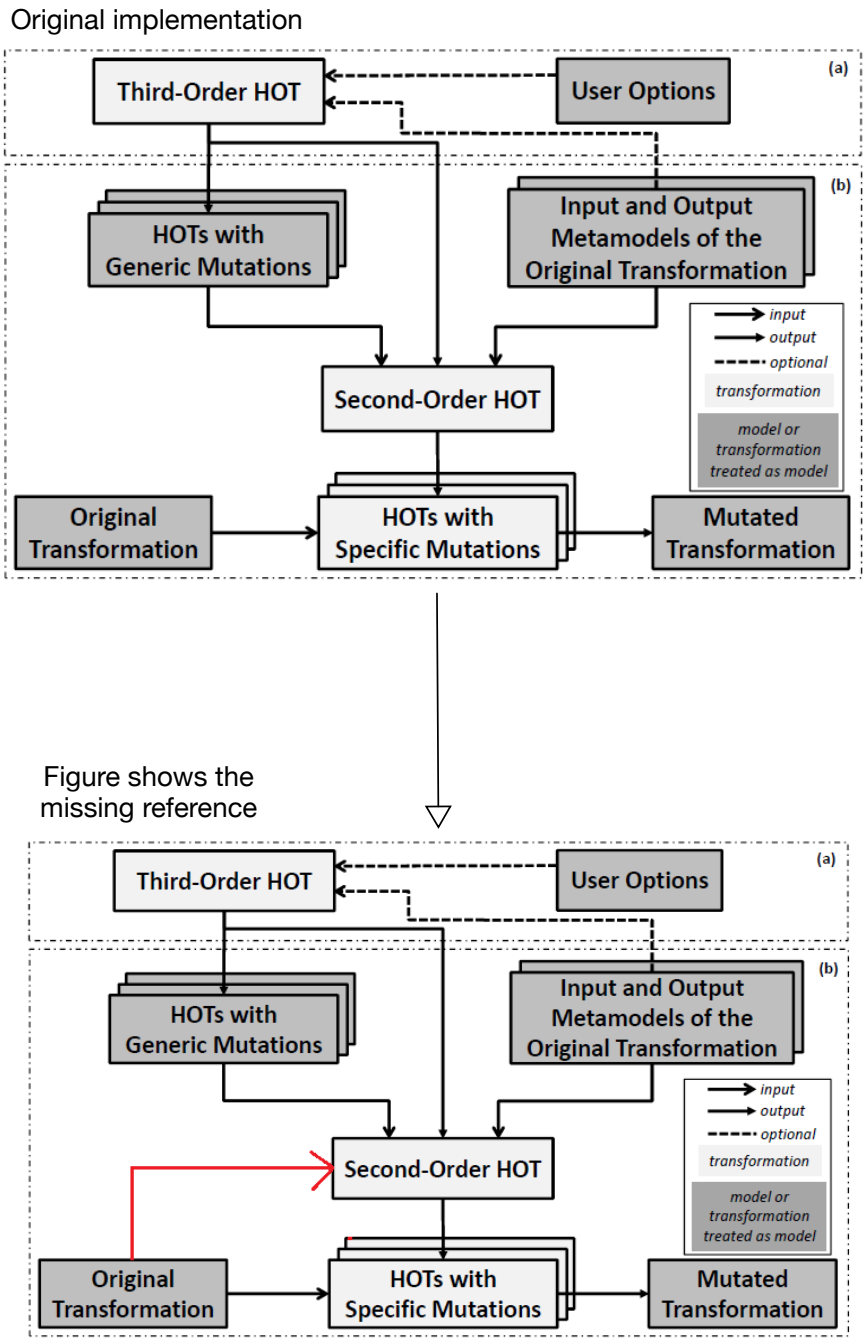

Fig. 9. Extension of the approach by Troya et al. [1]

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 bindingelements.

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 the value from the last binding for our new first element and the value from the first 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.

Listing 1.8. ValueChangeBinding-Definition using values from input models.

```
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
)

}
```

3.2 Implemented InPatternElement mutations

Deletion The deletion of an InPatternElement requires the handling of its usage in OutPatternElement bindings.

To delete an InPatternElement (IPE) 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. E.g. the rule *PNMLDocument* in listing 1.9 contains the IPE *e*. If it is deleted the usage *e* in the OutPatternElement has to be handled.

Listing 1.9. Rule with 2 existing IPEs

```
rule PNMLDocument {
    from
        e : PetriNet!PetriNet,
        f : PetriNet!PetriNet
```

```
5     to
6         n : PNML!PNMLDocument
7         (
8             location <- e.location,
9             xmlns <- uri,
10            nets <- net
11        )
```

Listing 1.10 shows what would happen if the OutPatternElement isn't changed too.

Listing 1.10. Rule with removed IPE

```
1  rule PNMLDocument {
2      from
3          f : PetriNet!PetriNet
4      to
5          n : PNML!PNMLDocument
6          (
7              location <- .location,
8              xmlns <- uri,
9              nets <- net
10         )
```

In case the IPE is deleted the corresponding binding will contain an unreferred variable. In this case the generated model is invalid. To prevent this from happening all unreferred variable bindings have to be deleted in a second step in the HOT.

A possible implementation of an IPE is shown in listing 1.11.

Listing 1.11. Delete IPE

```
1  rule DelteIPE {
2    from ipe : ATL!InPatternElement (
3      ipe.inPattern.elements -> size() > 1 and
4      ipe = ipe.inPattern.elements -> last()
5    )
```

Listing 1.12 shows another binding deletion.

Listing 1.12. Delete bindings after deleting an IPE

```
1  rule DelteNavigationBinding{
2  from b : ATL!Binding(
3      b.value.oclIsTypeOf(ATL!NavigationOrAttributeCallExp) and
4      b.value.source.referredVariable.oclIsUndefined()
5  )
6  to
7  }
8  rule DeleteCollectionBinding{
9  from b:ATL!Binding(
10     b.value.oclIsTypeOf(ATL!CollectionOperationCallExp) and
11     b.value.source.oclIsTypeOf(ATL!NavigationOrAttributeCallExp) and
12     b.value.source.source.referredVariable.oclIsUndefined()
13 )
14 to
15 }
```

To delete really all of the occurences of the IPE it is necessary to delete not only the simple binding itself, also more complex occurrences like in navigation expression (ATL!NavigationOrAttributeCallExp) or collection Expressions

(ATL!CollectionOperationCallExp). As these complex expressions can be encapsulated it is neccessary to do this deletion in some kind of recursion to cover really all occurrences.

4 Related work

In the related work [11] the authors have identified the mutations for ATL transformation language. We have implemented a set of possible mutations in ATL they have identified. For the general part of the paper we read some other papers with different approaches. In another referenced work [13] the authors have researched how it is possible to support mutation testing in the atlas transformation language. The authors have defined a set of mutation operators.

The autohors approach has been validated and the results have shown that their defined operators successfully detected insufficiency in the test suite.

5 Conclusion and future work

The overall goal of our work was to complement the approach by Troya et al. [1] by implementing a large set of mutations for ATL transformations. This implementation of mutation operators is discussed and evaluated regarding effectiveness. We have selected some mutations in ATL like Binding (Addtion, Deletion and Change), In Pattern Element (Addition, Class change) and Out Pattern Element (Deletion, Class change). In the paper we have explained the effect of the mutations of transformations. While working on this topic, we had to deal with 2 major problems:

- Consequences which occur when changing an element of a transformation.
- Needed input models for second-order HOTs.

Consequences of changing an tranformation element occured during the deleting of an IPE or OPE. If deleting such an element, also the bindings with references to these elements had to be deleted. The difficulty of determining such bindings was that the reference to the delete IPE or OPE could not be only a simple expression but also more complex expression such as IteratorExpressions, OperationCallExpressions or LoopExpressions. So all of the possibilities of references had to be considered. The second big problem was the amount of input models for second-order HOTs. We had to extend the overall approach with an additional input model - the original ATL transformation. This extension allowed us to define more complex mutations, such as the adding of a reference binding.

The approach presented in this paper aims at mutation of model transformations for building trust into model transformation programs using this technique. Further work will consist in development of tools for autmated mutations of model transformations. First only for ATL and second for other model transformation languages for example like QVT.

References

1. Bergmayr, A., Troya, J., and Wimmer, M. From out-place transformation evolution to in-place model patching. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2014), ASE '14, ACM, pp. 647–652.
2. Czarnecki, K., and Helsen, S. Classification of model transformation approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA* (Anaheim, CA, USA, 2003).
3. Frédéric Jouault, Freddy Allilaire, J. B., and Kurtev, I. Atl: A model transformation tool. *Software Engineering Group 37*, 5 (Aug 2007), 31–39.
4. Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on 37*, 5 (Sept 2011), 649–678.
5. Jordi Cabot. http://modeling-languages.com/introducing-higher-order-transformations-hots/. Accessed: 2015-04-15.
6. Mottu, J.-M., Baudry, B., and Le Traon, Y. Mutation analysis testing for model transformations. In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications* (Berlin, Heidelberg, 2006), ECMDA-FA'06, Springer-Verlag, pp. 376–390.
7. Myers, G. J., and Sandler, C. *The Art of Software Testing*. John Wiley & Sons, 2004.
8. Sendall, S., and Kozaczynski, W. Model transformation: The heart and soul of model-driven software development. *IEEE Softw. 20*, 5 (Sept. 2003), 42–45.
9. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. On the use of higher-order model transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications* (Berlin, Heidelberg, 2009), ECMDA-FA '09, Springer-Verlag, pp. 18–33.
10. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bzivin, J. On the use of higher-order model transformations. In *Model Driven Architecture - Foundations and Applications*, R. Paige, A. Hartman, and A. Rensink, Eds., vol. 5562 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 18–33.
11. Troya, J., Bergmayr, A., Burgueo, L., and Wimmer, M. Towards systematic mutations for and with atl. In *Proc. of the Eighth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 10th International Workshop on Mutation Analysis (Mutation 2015)* (2015).
12. Utting, M., Pretschner, A., and Legeard, B. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab. 22*, 5 (Aug. 2012), 297–312.
13. Yasser Khan, J. H. *Applying Mutation Testing to ATL Specifications: An Experimental Case Study*. IARIA, 2013.