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. Faults in model transformations will produce faults in 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 has to be correctly tested to maintain product quality. For that mutation testing is a popular technique. The mutation testing of model transformations have 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 solution of the implementation is 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: [16]

- Reverse engineering models e.g. in the process of replacing a legacy system . Then 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
- et cetera

Therefore the quality of the overall software solution is determined by the quality of the models and the resulting model transformations. [7] As a consequence testing the transformations for correctness is a essential part of the quality of the software. [19]

To check the quality of software is has to be tested. 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. [12] 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. [20]

Mutation testing or mutation analysis is a fault-based testing technique. It applies 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. It has been proven that mutation testing is useful as testing approach but also as: [8]

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. Due to the needed size of the sets it's to expensive to create them manually. Additionally the execution of the different models against the test data is a time-consuming.

Therefore automation is required for:

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

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

1.2 Contribution

Javier et. al. contributed three aspects.

1. They have defined a general languagecentric synthesis approach by defining a set of mutation operators based on ATL.
2. They made a concept and build a first version of a framework utilizing Higher order transformations for generating mutants. [18]
3. 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 he jargon of mutation testing the mutations are *killed*. Killing means recognizing differing results of the original system under test (SUT) and tested mutants. [11]

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.

The goal of the process**??** 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: [8]

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

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 [19].

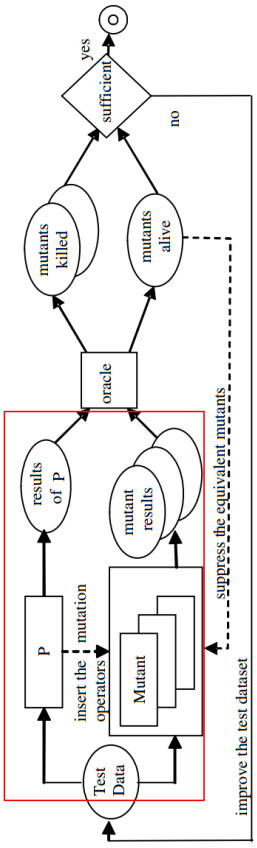

Fig. 1. The mutation testing workflow contains a feedback loop. The red marked area is the part this paper contributes to. [11]

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.

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. [4] 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 models from one or more input models. To successfully generate output-models it takes a clear understanding of syntax and semantic of the models and the relationship between their different entities. Model driven engineering provides a formal framework to define this prerequirements. [16]

At the high level exist two categories of model transformations: [3]

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 exist two different approaches to synthesize the source code. There exists 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 alternative to the visitor based approach is the template based appraoch. A template consists of constant text and parts which dynamically generate text from the input model. The dynamically part expands iterativly. [3]

There are multiple strategies for model to model transformations:

- *Direct-manipulation approache* offers a internal model representation and tools to manipulate it.
- The *relational approache* 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-transformations-based approache* utilizes graph theory, modelling model entities as nodes and edges in the graph, to define model transformations.
- The *structure-driven approache* provides the user with a framework to operate on a hierarchy to copy objects and their attributes.
- The combination of two or more concepts is called *hybrid approache*.

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 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. The transformation definition is a model conforming to the ATL metamodel. All metamodels conform to the MOF.

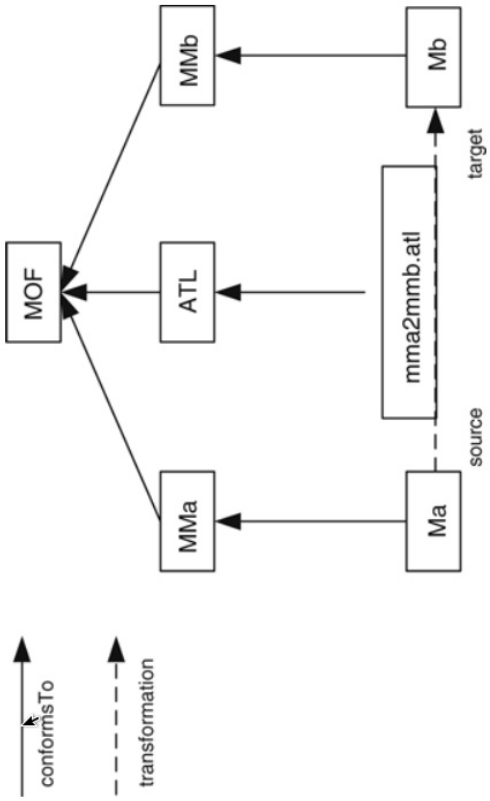

Fig. 2. Overview of the ATL transformational approach
[4]

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. [4]

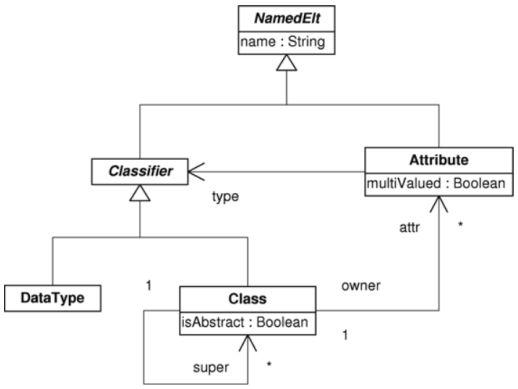

Fig. 3. Class metamodel
[4]

The class metamodel is used by the ATL transformation 1.1 shown in Listing 4 [4]

In this part we want to create a transformation program for a general example that transform simple class models to relational models. The source and the target model are presented in 3 and 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
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 mapped the elements EntityModel to FormModel.

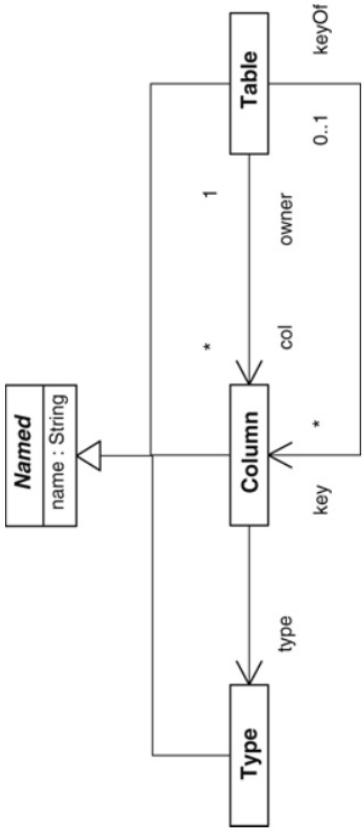

Fig. 4. Relational metamodel
[4]

Higher Order Transformations Higher order transformations are defined as a model transformation such that its input and/or output models are themselves transformation models. [17]

In the following figure a sample schema of a Higher order transformation for transformation modification in ATL is shown. The figure show a good overview for the model transformation such that its input/output models are themselves transformation models.

[9]

Therefore a transformation model is:

- The input of a Higher order transformation
- The output of a Higher order transformation
- Or it is both

Tisi et. al [17] 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** 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).

Another possibility of a Higher order transformation is a second Higher order transformation. That means that a Higher order transformation be also the input/output of another transformation.

2.3 Mutations in Model Transformations

In order to identify the possibility of mutations in languages we have to know about the concepts of the languages can be mutated. This mutation possibilities are presented in the referenced paper [19]. For a better overview in the following figure there are presented the possible mutations for ATL transformations.

In the column *Consequences* of there are some notices about the consequences of a mutation. In the referenced paper [19] the consequences descripted as follows: When a transformation is mutated, it has an effect in the output model that is generated for the same input model as with the original transformation.

There can be completely new objects that were not present before (OA: Object Addition). Some objects can be deleted (OD: Object Deletion). There can also be objects that are modified. They are labeled as OPI: Object Property Initialized or OPN: Object Property set to Null. That means that a property of

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]

an obejct is initialized or is set to null. Another possibility is when the value of a property of an object is modified (OPM: Object Property Modified). Or another case is that an object has replaced (OR: Object Replacement). For OR: Object Replacement it is important to know that Object Replacemnt can also be seen as the deletion and addition of an object. The relationships can also be added (RA: Relationship Added) or deleted (RD: Relationship Deleted). [19]

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. This are defined in 3 and show the current status of the work.

3.1 Implemented binding mutations

There are two types of major types of binding values in the ATL metamodel:

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

Table 2. The implementation goals

– 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 figure 6)

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

Deletion Deleting a Binding is the implicitly omitting of a 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 }
```

Addition The addition 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 is missing.

Troya et al propused in their paper [1] second order HOTs which are applied to the result of the first HOTs. The result of the first HOTs transformations contains 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 actual available one.

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 propused algorithm is showed by listing 1.3

Listing 1.3. AddBinding-Definition

```
rule AddBinding{
  from
  ope : ATL!OutPatternElement (
    ope.bindings -> forAll( b | b.propertyName <> 'NewBindingPropertyName'
        )
  )
  to
  ope2 : ATL!OutPatternElement (
    bindings <- ope.bindings -> append(bindingNewElement)
  ),
  bindingNewElement : ATL!Binding (
    outPatternElement <- ope2,
    propertyName <- 'NewBindingPropertyName',
    value <- newValue
  ),
  newValue : ATL!StringExp (
    stringSymbol <- 'testvalue'
  )
}
```

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

Listing 1.4. AddBindingNames-Definition.

```
rule AddBindingNames {
from b : ATL!StringExp (
    b.stringSymbol = 'NewBindingPropertyName'
)
using{
  classes : Sequence(OUT_MM!EClass)
    = OUT_MM!EClass.allInstancesFrom('OUT_MM')
    -> select(c | c.getESuperTypes() -> size() <= 0);
}
to b2 : ATL!StringExp (
  stringSymbol <- classes
    -> first().getEStructuralFeatures() -> last().name
)
}
```

Value change The simplest value change mutation is changing a primitive expression. The example in listing 1.5 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 1.5. ValueChangeBinding-Definition.

```
rule ValueChangeBinding_All{
    from
     b : ATL!Binding (b.value.oclIsTypeOf(ATL!StringExp))
    to
     c : ATL!Binding (
        value <- newStringExp
     ),
     newStringExp : ATL!StringExp(
        stringSymbol <- 'hello'
     )
}
```

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 1.6 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 1.6. ValueChangeBinding-Definition using same constant string value.

```
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.

Listing 1.7. 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 (
```

```
10      bindings <- ope.bindings -> excluding (ope.bindings -> first())
11          -> excluding (ope.bindings -> last())
12          -> prepend (bindingNewFirst)
13          -> append (bindingNewLast)
14  ),
15  bindingNewLast : ATL!Binding (
16      outPatternElement <- ope2,
17      propertyName <- (ope.bindings -> last()).propertyName,
18      value <- (ope.bindings -> first()).value
19  ),
20  bindingNewFirst : ATL!Binding (
21      outPatternElement <- ope2,
22      propertyName <- (ope.bindings -> first()).propertyName,
23      value <- (ope.bindings -> last()).value
24  )
25
26  }
```

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 3.2 contains the IPE *e*. If it is deleted the usage *e* in the OutPatternElement has to be handled.

```
1  rule PNMLDocument {
2      from
3          e : PetriNet!PetriNet,
4          f : PetriNet!PetriNet
5      to
6          n : PNML!PNMLDocument
7          (
8              location <- e.location,
9              xmlns <- uri,
10             nets <- net
11         )
```

Listing3.2 shows what would happen if the OutPatternElement isn't changed too.

```
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. This would void the generated model. 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 3.2.

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

Listing 3.2 shows another binding deletion.

```
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 }
```

4 Related work

In the related work [19] the authors has identified the mutations for ATL transformation language as in chapter *Mutations in Model Transformations* descripted. We have implemented a set of possible mutations in AATl they have identified. For the general part of the paper we have read some other papers with different approaches. In another referenced work [22] the authors has researched how it is possible to support mutation testing in the atlas transformation language. The authors has defined a set of mutation operators. This operators are:

- Matched to Lazy (M2L)
- Lazy to Matched (L2M)
- Delete Attribute Mapping (DAM)
- Add Attribute Mapping (AAM)
- Delete Filtering Expression (DFE)
- Add Filtering Expression (AFE)
- Change Source Type (CST)
- Change Target Type (CTT)
- Delete Return Statement (DRS)
- Delete Use Statement (DUS)
- Change Execution Mode (CEM)

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 broader goal of this work is to create an introduction to a specific field of model driven engineering (MDE). Integral part of MDE is model transformation. [16]. In this paper we have implemented some solutions regarding mutations in ATL. The implementation of a set 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 explain the effect of the mutations of transformtions and we see that there are many consequences. 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 deveolpment of tools for autmated mutations of model transformations. First only for ATL and second for other model transformation languages for example like QVT.

6 Bibliographic Issues

6.1 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`.

6.2 BibTeX

BibTeX should be used for referencing.

The LaTeX source document of this pdf document provides you with different samples for references to journals [6], conference papers [14], books [5], book chapters [15], electronic standards [13], dissertations [21], masters' theses [10], and web sites [2]. 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.

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. Business Informatics Group. http://www.big.tuwien.ac.at. Accessed: 2010-11-09.
3. 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).

4. Frédéric Jouault, Freddy Allilaire, J. B., and Kurtev, I. Atl: A model transformation tool. *Software Engineering Group 37*, 5 (Aug 2007), 31–39.
5. Hitz, M., Kappel, G., Kapsammer, E., and Retschitzegger, W. *UML @ Work, Objektorientierte Modellierung mit UML 2*, 3. ed. dpunkt.verlag, 2005 (in German).
6. Huemer, C., Liegl, P., Schuster, R., and Zapletal, M. B2B Services: Worksheet-Driven Development of Modeling Artifacts and Code. *Computer Journal 52*, 2 (2009), 28–67.
7. Hutchinson, J., Rouncefield, M., and Whittle, J. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 633–642.
8. 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.
9. Jordi Cabot. http://modeling-languages.com/introducing-higher-order-transformations-hots/. Accessed: 2015-04-15.
10. Langer, P. Konflikterkennung in der Modellversionierung. Master's thesis, Vienna University of Technology, 2009.
11. 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.
12. Myers, G. J., and Sandler, C. *The Art of Software Testing.* John Wiley & Sons, 2004.
13. OASIS. *Business Process Execution Language 2.0 (WS-BPEL 2.0)*, 2007.
14. Schauerhuber, A., Wimmer, M., Schwinger, W., Kapsammer, E., and Retschitzegger, W. Aspect-Oriented Modeling of Ubiquitous Web Applications: The aspectWebML Approach. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07), March 26-29, Tucson, Arizona, USA* (2007), IEEE CS Press, pp. 569–576.
15. Schwinger, W., and Koch, N. Modeling Web Applications. In *Web Engineering*, G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger, Eds. John Wiley & Sons, Ltd, 2006, pp. 39–64.
16. Sendall, S., and Kozaczynski, W. Model transformation: The heart and soul of model-driven software development. *IEEE Softw. 20*, 5 (Sept. 2003), 42–45.
17. 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.
18. 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.
19. 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).
20. Utting, M., Pretschner, A., and Legeard, B. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab. 22*, 5 (Aug. 2012), 297–312.

21. Wimmer, M. *From Mining to Mapping and Roundtrip Transformations - A Systematic Approach to Model-based Tool Integration.* PhD thesis, Vienna University of Technology, 2008.
22. Yasser Khan, J. H. *Applying Mutation Testing to ATL Specifications: An Experimental Case Study.* IARIA, 2013.

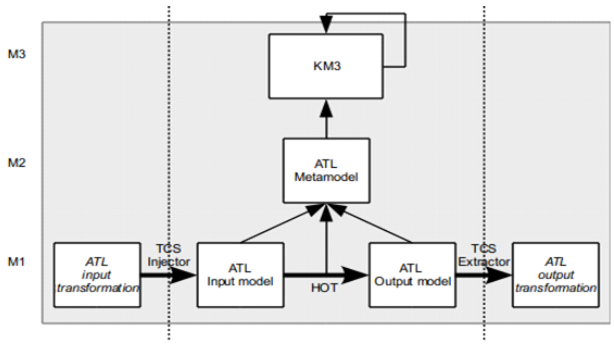

Fig. 5. Sample schema of a Higher Order Transformations for transformation modification in ATL

[9]

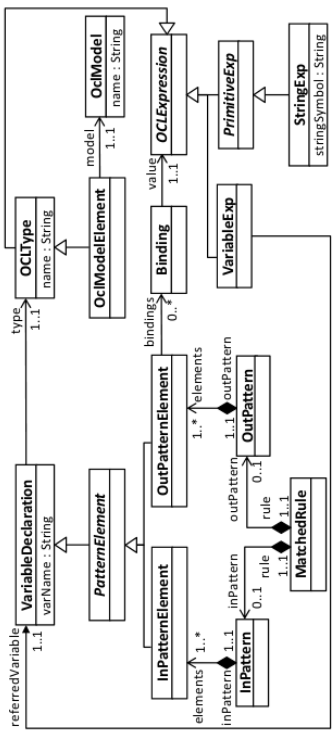

Fig. 6. Excerpt of the ATL metamodel