Towards Systematic Mutations for and with ATL Model Transformations

Javier Troya*, Alexander Bergmayr*, Loli Burgueño†, and Manuel Wimmer*
*Business Informatics Group, Vienna University of Technology, Austria
Email: {troya,bergmayr,wimmer}@big.tuwien.ac.at
†GISUM/Atenea Research Group, Universidad de Málaga, Spain
Email: loli@lcc.uma.es

***Abstract*—Model transformation is a key technique to automate software engineering tasks, such as generating implementations of software systems from higher-level models. To enable this automation, transformation engines are used to synthesize various types of software artifacts from models, where the rules according to which these artifacts are generated are implemented by means of dedicated model transformation languages. Hence, the quality of the generated software artifacts depends on the quality of the transformation rules applied to generate them. Thus, there is the need for approaches to certify their behavior for a selected set of test models. As mutation analysis has proven useful as a practical testing approach, we propose a set of mutation operators for the ATLAS Transformation Language (ATL) derived by a comprehensive language-centric synthesis approach. We describe the rationale behind each of the mutation operators and propose an automated process to generate mutants for ATL transformations based on a combination of generic mutation operators and higher-order transformations. Finally, we describe a cost-effective solution for executing the obtained mutants.**

Keywords*—*Mutation, Model Transformations, ATL, Higher-Order Transformations

I. Introduction

Model transformation is a key technique to automate software engineering tasks in Model-Driven Engineering (MDE) [1], [2]. To enable this automation, transformation engines are available which are able to synthesize various types of software artifacts from models, where the rules according to which these artifacts are generated are implemented by means of dedicated model transformation languages. Hence, the quality of the generated software artifacts is highly affected by the quality of the developed model transformations. As a result, model transformations are subject to thorough tests for correctness [3]–[6]. For this reason, several approaches have been proposed that either verify the correct behavior of the transformations using formal methods [7] or certify their behavior for a selected set of test models mainly to identity bugs in a cost-effective way [8], [9].

As mutation analysis has proven to be useful as a practical testing approach [10], it is also applied to test model transformations mainly for the generation of (i) test data in terms of input models [11], [12] and (ii) mutants of model transformations [13]–[15]. Considering the latter approaches, they propose generic mutation operators, i.e., independent of a particular transformation language, and mutation operators that are dedicated to a specific language, such as the ATLAS Transformation Language (ATL) [16], [17]. As a result, the focus of these approaches is mainly set on partially identifying effective mutation operators while neglecting means to automate the generation of mutated model transformations and to efficiently execute them.

Problem. Due to the fact that mutation analysis requires a complete set of mutation operators, and consequently, a large number of mutated model transformations to be effective, manually generating them appears infeasible [18]. Moreover, their execution against the input models leads typically to high computational costs [10], particularly if a mutant is considered as a complete re-execution of the original transformation to which a fault is injected. Hence, automation is required to cope with the challenges of generating an effective set of mutation operators as well as mutated model transformations. Furthermore, to deal with the execution of mutants, techniques are required to reduce the computational costs of executing the model transformation mutants.

Contribution. Our contribution is threefold: (i) we have derived a systematic set of mutation operators dedicated to ATL by proposing a general language-centric synthesis approach for mutation operators, (ii) we have automated the generation of model transformation mutants by realizing a framework that exploits the concept of Higher-Order Transformations (HOTs) [19], and (iii) we have integrated into our framework means to reduce the computational costs of executing model transformation mutants by relying on techniques for incremental model transformation execution, which we have proposed in previous work [20].

Structure. In Section II, we give the background for our work by introducing ATL by-example. We provide details regarding our mutation operator synthesis approach and the ATL mutation operators in particular in Section III. Then, in Section IV, we present our framework for automating the generation of mutated ATL model transformations and give insights into how they can be executed in a cost-effective way. In Section V, we introduce the prototypical implementation of our proposed framework and apply it to the example introduced in Section II. Finally, we discuss related work in Section VI before we conclude with an outlook on future work in Section VII.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th International Workshop on Mutation Analysis (Mutation 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

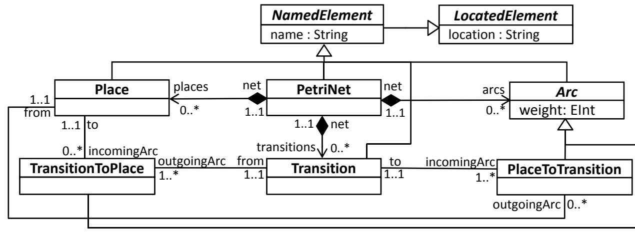


Fig. 1. PetriNet Metamodel.

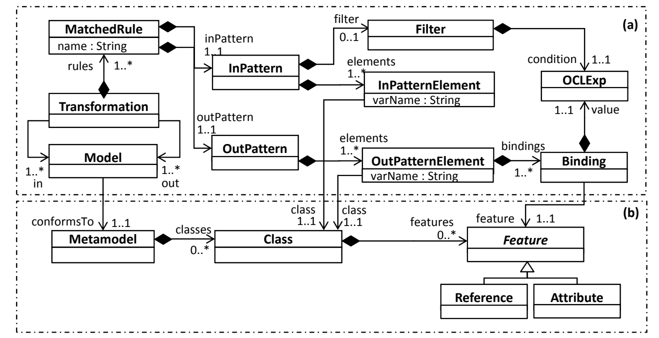


Fig. 2. Metamodel excerpts: (a) ATL transformation language and (b) metamodeling language.

II. Model Transformations in MDE

In this paper we focus on ATL, which is shortly introduced in this section.

A. ATL

ATL is a hybrid model transformation language containing a mixture of declarative and imperative constructs—in this paper we focus only on its declarative part. Both out-place and in-place transformations can be defined in ATL. In an out-place transformation, a new output model is created from the input one. The transformation specifies which concepts of the output model are created from which ones of the input model. The *default mode* of ATL is used for this. In-place rules are defined using the *refining mode* of ATL. In the refining mode, the input model evolves to obtain the output one. Consequently, the transformation specifies how the input model has to change in order to obtain the output one.

Listing 1 displays an excerpt of an out-place transformation that generates a PNML (Petri Net Markup Language) model from a PetriNet model. It has been taken from the ATL Zoo[1] and will be referenced throughout the paper to explain our approach. The PetriNet metamodel, to which input models for this transformation conform, is displayed in Figure 1.

Figure 2(a) shows the structure of the declarative part of an ATL *Transformation*, which is composed of declarative *MatchedRules* (lines 4, 14 and 24 in Listing 1). It receives *Models* as input and produces output *Models*, which conform to a metamodel (cf. Figure 2(b) and line 2 in the listing). A *MatchedRule* contains one *InPattern* (starting with the keyword *from*, cf. line 5) and one *OutPattern* (starting with the keyword *to*, cf. line 7). The former is a query on the input model and gathers the set of *InPatternElements* (line 6 for instance) that represent the input model elements of the rule. It can also contain a *Filter* (none is specified in this transformation). If the conditions of such a *Filter* are satisfied by the *InPatternElements*, the respective rule is applied. *Filters* are specified by means of OCL (Object Constraint Language) expressions. *OutPatterns* describe the creation of elements in the output model. Such elements are of type *OutPatternElements* (e.g., the one defined in line 8). Each *OutPatternElement* is composed of a set of *Bindings* (lines 9, 10 and 11). Their values are expressed and computed by OCL expressions that are used to initialize the features of output model elements.

ATL, as other transformation languages, has a feature that we call *inter-rule dependencies*. In our example, the *from* of the value expression *e."from"* (line 36) is either of type *Transition* or *Place* (cf. PetriNet metamodel in Figure 1), depending on if *e* is an *Arc* of type *TransitionToPlace* or *PlaceToTransition*, respectively. For that reason, when the *Binding* is computed, *source* will reference those elements created either in rule *Transition* or *Place*. To do so, ATL performs a transparent lookup of output model elements for given input model elements when executing *Bindings*. Hence, it automatically retrieves the corresponding *PNML!Transition* and *PNML!Place* elements for the queried *PetriNet!Transition* and *PetriNet!Place* elements, respectively.

Listing 1. *PetriNet2PNML* ATL Transformation.

```
 1 module PetriNet2PNML;
 2 create OUT : PNML from IN : PetriNet;
 3
 4 rule Place {
 5   from
 6     e : PetriNet!Place
 7   to
 8     n : PNML!Place (
 9             name <- name,
10             id <- e.name,
11             location <- e.location),
12     name : PNML!Name (labels <- label),
13     label : PNML!Label (text <- e.name)
14 }
15
16 rule Transition {
17   from
18     e : PetriNet!Transition
19   to
20     n : PNML!Transition (
21             name <- name,
22             id <- e.name,
23             location <- e.location),
24     name : PNML!Name (labels <- label),
25     label : PNML!Label (text <- e.name)
26 }
27
28 rule Arc {
29   from
30     e : PetriNet!Arc
31   to
32     n : PNML!Arc (
33             name <- name,
34             location <- e.location,
35             id <- e.name,
36             source <- e."from",
37             target <- e."to"),
38     name : PNML!Name (labels <- label),
39     label : PNML!Label (text <- e.name)
40 }
```

B. Higher-Order Transformations

A *Higher-Order Transformation* (HOT) [19] is a model transformation such that its input and/or output models are

[1] http://www.eclipse.org/atl/atlTransformations/

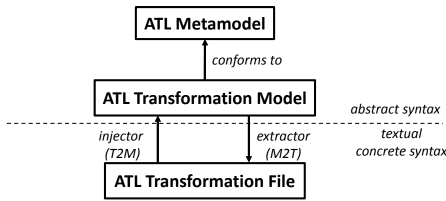


Fig. 3. Injections and Extractions of ATL Transformations.

themselves transformation models. A transformation model is a model transformation that is represented as a model, instead of being represented with its textual syntax. The metamodel to which such models conform in our case is the ATL metamodel. In order to obtain a model representation of a transformation file, we apply a text-to-model transformation called *injector* (cf. Fig 3). On the other hand, when we want to obtain a textual representation of a transformation model, we apply a model-to-text transformation called *extractor*. Please note that a HOT may be also the input/output of another transformation, thus there are second-order HOTs and so on. These concepts are important for the explanation of our approach in Section IV.

III. Mutations in Model Transformations

In order to define mutations for a language, we first need to know which concepts of the language can be mutated. In this work we have applied a systematic analysis based on the concepts described in the ATL metamodel for identifying mutation operators, what is presented in Section III-B. Each of the mutations can influence in different ways the output models generated by the mutants. For this reason, we first analyze the different consequences that a mutation may have in Section III-A. Finally, in Section III-C, we describe our approach to obtain the dependencies among rules in an ATL transformation, what can be useful in the testing process.

A. Consequences of a Mutation

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. The alterations that may take place in the output model are graphically depicted in Figure 4. In the explanation of the consequences that mutations involve, we will be comparing the output model obtained by the original transformation (such as the model in Figure 4(a)) with the one obtained by a mutation of the transformation (such as the model in Figure 4(b)).

To begin with, there can be completely new objects that were not present before (OA: *Object Addition*). Likewise, some objects may be deleted (OD: *Object Deletion*). There can also be objects that are modified. This means that the object is the same, but some of its properties (attributes and/or references) have been modified. This happens for instance when a property of the object that was set to null is now initialized (OPI: *Object Property Initialized*), or, on the contrary, a property that had a value is now set to null (OPN: *Object Property set to Null*). An object can also be modified when the value of one property is modified (OPM: *Object Property Modified*), think for instance of the string attribute *name* of an object. It could also be the case that an object of a certain type has replaced the object of a different type (OR: *Object Replacement*), meaning that

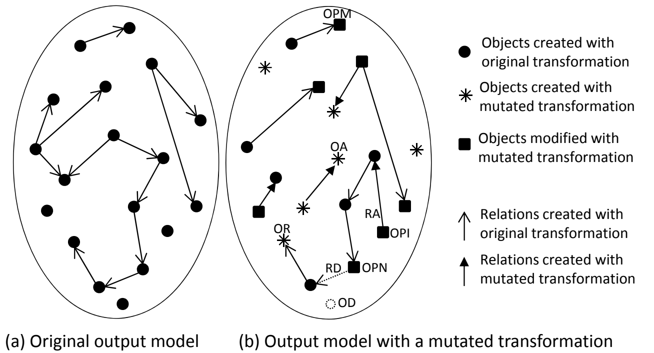


Fig. 4. Possible alterations in output model (an object in the same position means it is the same object).

the latter keeps the incoming and outgoing relationships of the former. In such a case, and in order to comply the conformance relationship to the output metamodel, the classes to which both objects conform must have an inheritance relationship in the metamodel, so that they share the types of their input and output relationships. Note that OR can also be seen as the deletion and addition of an object.

Regarding relationships among objects, they can also be added (RA: *Relationship Added*) or deleted (RD: *Relationship Deleted*). For instance, the modification of an object may imply the creation/deletion of one of its relationships. Another example is when a new object is created and its relationships are initialized. On the contrary, when an object is deleted, so are the relationships that start from it or end in it.

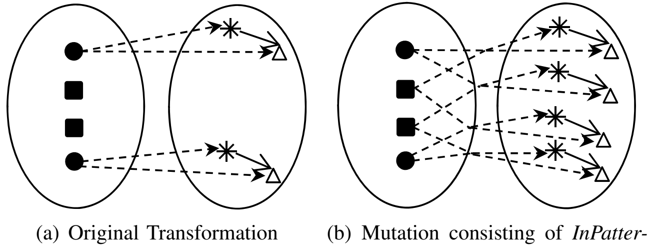
B. Mutation Possibilities

According to the excerpt of the ATL metamodel shown in Figure 2(a), we identify a set of possible mutations. We aim for completeness of our approach by systematically considering the addition and deletion of instances for any metaclass in the transformation metamodel and modifications of their features.

Table I shows the set of mutations identified, and the consequences they imply in the generated output model (cf. Section III-A), where a consequence enclosed within square

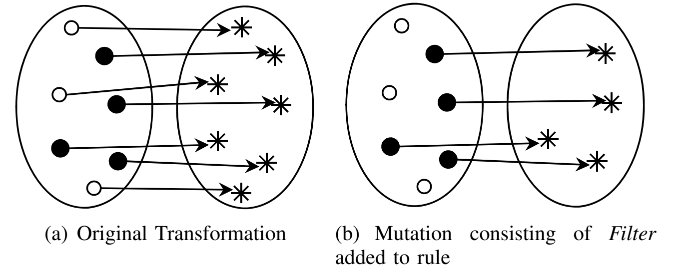
TABLE I. Mutations Identified for ATL Transformations.

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	OPI;[RA] OPN;[RD] OPM;[RA];[RD]



(a) Original Transformation (b) Mutation consisting of *InPatternElement* added to rule

Fig. 5. Effect of Adding *InPatternElement*.



(a) Original Transformation (b) Mutation consisting of *Filter* added to rule

Fig. 6. Effect of Adding *Filter*.

brackets means that it may happen or not. In the following explanation, we compare the output model that is created by the original transformation with the one that would be created by the transformation where the specific mutations are applied.

MatchedRule. Adding or deleting a *MatchedRule* implies to add or delete the objects that the rule creates, and the possible relationships among them.

InPatternElement. If an *InPatternElement* is added or deleted, the matches of a rule for a given input model may change as well. For instance, if we had only one *InPatternElement* in a rule and we add another one, the match is realized now with the cartesian product of both element types, so that more elements are created. Think for instance of a rule that has one *InPatternElement* to realize matches with (objects of type circle, Figure 5(a)), and the rule produces two *OutPatternElements* and a relationship among them. Then, a mutation consists of adding a new *InPatternElement* of type square to the rule, so that now the matching is produced with the cartesian products of circles and squares (Figure 5(b)), producing more elements in the output model. Contrarily, if we remove an *InPatternElement*, the number of matches for a rule may decrease. Furthermore, the addition or deletion of an *InPatternElement* may lead to a change of the *OutPattern* in the rule provided that the variable referring to the new/old *InPatternElement* is used in one or several *Bindings*. Similarly, when the *class* feature of an *InPatternElement* changes, we consider it as a deletion and an addition of an *InPatternElement*.

Filter. The effect of the addition, deletion and modification of a *Filter* is connected to the number of objects that match the rule. When we add a *Filter*, we make the rule application more restrictive, since the objects matching the rule need to satisfy a certain property. Consider for instance a rule that creates an object from each object of type circle (cf. Figure 6(a)). Then, we add a *Filter* to only transform those circles that are filled, as shown in Figure 6(b), so that less elements are created. The deletion of a *Filter* has the contrary effect. Finally, if the condition of a *Filter* is changed, it implies the same as deleting it and adding it again. This means that some objects will be matched that previously were not, and the opposite. Consequently, new objects are created in the output model and some others are deleted.

OutPatternElement. If a new *OutPatternElement* is created in a rule, then we have new objects in the output model, and possibly new relationships. When one is deleted, then we have less objects, where the relationships starting and ending in these objects are also deleted. Changing the *class* attribute of an *OutPatternElement* means that the previously created object is replaced by one of a different type, and so relationships can also be created and deleted.

Binding. When a *Binding* is added, it means that the value of a property is initialized. If such a property is a relationship, then it is created. When a *Binding* is deleted, the value of the property is set to null, since no value is given to it. When its *value* is modified, it can mean two things depending on what the property is. On the one hand, if it is an attribute, then its value changes (think for instance of the modification of a string value). On the other hand, if it is a relationship, it means that the previous relationship is deleted and a new one is created.

C. Dependencies among Rules

When a certain rule is mutated in a model transformation, it has a direct effect on the elements that are created by such a rule. However, at the same time, mutations can also have consequences on the elements produced by other rules, provided these have inter-rule dependencies (cf. Section II-A) with the mutated one. In can be interesting in the process of testing model transformation to know if rules have dependencies with mutated rules. For this reason, we propose an approach to obtain the dependencies among ATL rules.

We compute the dependencies among rules with a HOT that takes as input the transformation injected into a model-based representation (cf. Section II-B) as well as the input metamodel of the transformation, from where we can statically infer information about types in the transformation. The HOT adds information to the transformation by initializing derived attributes that we have added to the *MatchedRule* class of the ATL metamodel, as shown in Figure 7. The transformation consists of two steps. First, the types of the rules are statically extracted, i.e., the classes of the input metamodel that participate in the rules. Second, these types are used to compute the dependencies.

Types Extraction. As explained in Section II-A, ATL performs a transparent lookup when objects created in a rule need to establish relationships with objects created in another rule. More specifically, if the type retrieved by the OCL expression of a *Binding* (*value* feature) in rule R1 is the same as the type

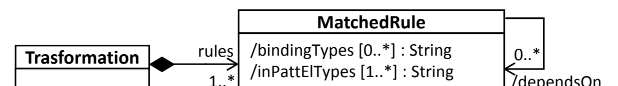


Fig. 7. ATL extension for considering dependencies.

of an *InPatternElement* in rule R2, then R1 depends on R2. Since ATL does not offer any support nor API to statically obtain the types of the elements appearing in the rules, the process is not trivial when OCL expressions play part of it.

Obtaining the types of *InPatternElements* is rather straightforward, since we only need to access the *class* relationship (Figure 2). The most challenging part is to extract the types from the OCL expressions of the *Bindings*. OCL expressions are textual expressions built conforming to the OCL package of the ATL metamodel, and their types extraction is specially challenging when they involve collection operators (*collect*, *select*, *reject*, etc.). Let us recall that the purpose of an OCL expression in a model transformation is to retrieve an object, or a collection of objects, that need to be accessed by means of a navigation through other elements in the model. In OCL, this is expressed by navigating through the references in a metamodel level. In order to extract the type of an OCL expression, we extract the type of the eventual element that is reached through the navigation.

Dependencies Computation. After we have extracted the types of the *InPatternElements* of each rule as well as the types appearing in the *Bindings*, we can easily calculate the dependencies among them. Thus, we consider that a rule, R1, *depends on* another rule, R2, if the intersection of the types of the *Bindings* of R1 with the ones of the *InPatternElements* of R2 is not empty. The generated model contains information about the dependencies among rules (*dependsOn* relationship, Fig. 7). As for the attributes, *bindingTypes* and *inPattElTypes* contain sequences of the types present in the *Bindings* and *InPatternElements*, respectively.

IV. Approach to Automate Mutations

We now present our approach for automating the generation of model transformation mutants (Sections IV-A and IV-B) and the means to reduce the computational costs of executing mutants (Section IV-C).

A. Automation through HOTs

Our approach is summarized in Figure 8. It is based on the use of HOTs. We focus for now on the lower part, Figure 8(b), and describe its components in the following.

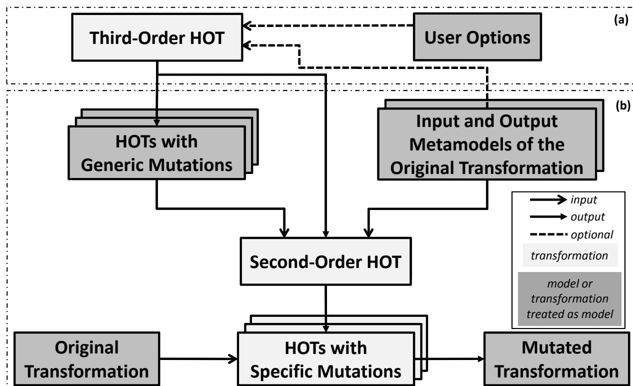


Fig. 8. Automating Mutations on Transformations.

1) HOTs with Generic Mutations: In order to automate the mutation of model transformations, we need to come up with an approach in which *generic mutations* are considered. We say that a mutation is generic when it has not been defined in the context of a specific transformation. The idea is that these generic mutations can then be applied to mutate any model transformation.

In order to identify which kind of mutations we can define, we refer at this stage to Table I (more mutations, exploring more deeply the ATL metamodel and its OCL package, are to be identified, categorized and implemented as future work). The idea is to have a set of HOTs with generic mutations (cf. upper-left box in Figure 8(b)). In fact, this set is extensible in the sense that new generic mutations can be implemented. As an example, consider the transformation shown in Listing 2, aimed at performing the mutation *Addition of InPatternElement*.

Listing 2. Gen Mutation *Addition of InPatternElement*.

```
1  -- @atlcompiler emftvm
2
3  module AddInPatternElement;
4  create OUT : ATL refining IN : ATL;
5
6  helper def : varNames : Sequence(String) = Sequence{
7    'a', 'aa','b','bb','c','cc','d','dd', '...'};
8
9  rule AddInPatternElement {
10  from
11   s : ATL!InPattern
12    (ATL!Rule.allInstances()->first() = s."rule")
13  to
14   t : ATL!InPattern (
15     elements <- s.elements -> append(newIPE)),
16   newIPE : ATL!InPatternElement (
17     varName <- thisModule.varNames->any(n |
18       ATL!PatternElement.allInstances()->
19       collect(pe|pe.varName)->excludes(n)),
20     type <- newOCLType),
21   newOCLType : ATL!OclModelElement(
22     model <- s.elements->first().type.model,
23     name <- 'Complete_IPE' )
24 }
```

Before explaining the transformation, let us recall that this transformation will be the input of a second-order HOT (box in the center of Figure 8(b)) that transforms it into a HOT with a specific mutation for a specific ATL transformation. This means that, later in the process, the transformation obtained out of this one will be used to actually mutate an ATL transformation.

This HOT with the generic mutation is, consequently, an in-place transformation. It takes an ATL transformation as input and modifies it. In particular, we can see in line 11 that the rule *AddInPatternElement* is taking an element of type *InPattern* as input, which is going to be modified by adding an *InPatternElement* to its *elements* and initializing its features (such modification begins in line 14). In particular, the rule matches the *InPattern* of the first rule defined in the transformation given as input, as specified in the *Filter* in line 12. To better understand the following explanation, the reader is referred to the excerpt of the ATL metamodel (combining features of the ATL and OCL packages) shown in Figure 9.

Line 15 shows that a new object, identified as *newIPE*, is added to the *elements* of the *InPattern*. This new object is of type *InPatternElement*, as we can see when it is created in

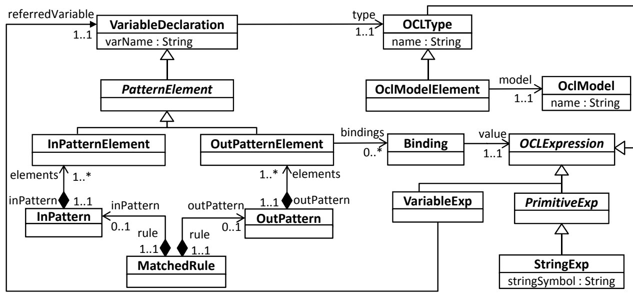


Fig. 9. Excerpt of ATL MM (with OCL Package).

line 16. Two features have to be defined for the newly created *InPatternElement*: its *varName* and its *type*.

The former, *varName*, is an attribute that keeps the name used to identify the *InPatternElement*, so it has to be unique. For this reason, the transformation checks (lines 17-19) that the variable name that is going to be given to it, which is a name included in the sequence *varNames* defined in the helper of line 6—please note that this sequence may contain as many different strings as needed—, is not already used in the transformation. For this reason, the transformation takes, using the *any* operation, an element of the sequence *varNames* that is not included in the names of the variables already associated to the *PatternElements* of the transformation that this HOT takes as input.

Listing 3. Gen Mutation *Deletion of OutPatElement*.

```
1  -- @atlcompiler emftvm
2
3  module DeleteOutPatternElement;
4  create OUT : ATL refining IN : ATL;
5
6  rule DeleteOutPatternElement {
7   from
8    ope : ATL!OutPatternElement
9    (ope.outPattern.elements->size() > 1 and
10    ope = ope.outPattern.elements -> last() and
11    ope.outPattern.rule =
12      ope.outPattern.rule.module.elements -> last())
13  to
14 }
15
16 rule DeleteAssociatedBinding {
17  from
18   b:ATL!Binding(b.value.oclIsTypeOf(ATL!VariableExp)
19      and b.value.referredVariable.oclIsUndefined())
20  to
21 }
```

As for the second feature of the *InPatternElement*, *type*, it is a relationship pointing an object of type *OclModelElement*. For this reason, a new object of this type, identified as *newOCLType*, is created (line 21). Two features need to be defined for the *OclModelElement*. One is a reference to the *model* to which the created *InPatternElement* belongs, while the other one is the *name* of the class that it represents. The *model* reference (line 22) will point to the same model to which the first already existing *InPatternElement* is pointing (this will be materialized when the HOT with the specific mutation is executed). As for the *name* attribute (line 23), since there is no information at this point about the classes of the input metamodel of the transformation that the user wants to mutate (recall that we are defining the mutation in a generic way), we simply write *'Complete_IPE'*, indicating that the information about the specific class for the *InPatternElement* needs to be added when executing the second-order HOT.

Another example of a HOT with a generic mutation is shown in Listing 3. Its purpose is to delete an *OutPatternElement* and all the *Bindings* that point to it. The deletion of the *OutPatternElement* is done in the first rule, namely *DeleteOutPatternElement*. Due to the *Filter*, starting in line 9, the left-hand side of the rule matches the last *OutPatternElement* (line 10) of the last rule of the transformation that is the input for this HOT (line 11), provided that there are more than one *OutPatternElements* (line 9). Then, there is nothing in the right-hand side of the rule, producing the deletion of the matched element, as explained in Section II-A.

There may be *Bindings* in different *OutPatternElements* of a transformation that refer to an object that is created through an *OutPatternElement* in the same rule. See for instance lines 38 and 39 in Listing 1. In this case, the *value* feature of the *Binding* in line 38 is of type *VariableExp* (cf. Figure 9). For this reason, in order to avoid having *Bindings* pointing null in the mutated transformation, we also delete those *Bindings* that were pointing to the deleted *OutPatternElement*. This is done by rule *DeleteAssociatedBinding*, which matches those *Bindings* (line 18) that should be pointing to an *OutPatternElement* but are actually pointing to null (the *VariableExp* class mentioned before has a pointer, *referredVariable*, to the actual *OutPatternElement*). The matched *Bindings* are then deleted (line 20).

2) Second-Order HOT: The *Second-Order HOT* shown in Figure 8(b) transforms the *HOTs with Generic Mutations* into *HOTs with Specific Mutations*. In Listing 4 we show an example rule that deals with making specific the generic rule shown in Listing 2.

Listing 4. Second-Order HOT.

```
1  -- @atlcompiler emftvm
2
3  module SecondOrderHOT;
4  create OUT : ATL refining IN : ATL, IN_MM : IN_MM,
5                           OUT_MM : OUT_MM;
6
7  helper def : random() : Real = "#native"!"java::util
8    ::Random".newInstance().nextDouble();
9
10 rule CompleteAddInPatternElement {
11  from
12   s : ATL!StringExp (s.stringSymbol = 'Complete_IPE')
13  using {
14   classes : Sequence(IN_MM!EClass) = IN_MM!EClass.all
15    InstancesFrom('IN_MM')->select(c|not c.abstract);}
16  to
17   t : ATL!StringExp (
18        stringSymbol <- classes->at((thisModule.
19         random()*classes->size()).floor()+1).name
20 }
```

To begin with, we can see that this in-place transformation has more inputs other than the transformation that is to be refined, *IN : ATL*. Thus, an in-place transformation may contain inputs that are used only for navigation purposes, but are not going to be modified. In this case, they are the input and output metamodels of the transformation that will be eventually mutated (represented by *IN_MM* and *OUT_MM*, respectively). We also need to explain that when there is a *Binding* of the form *feature* <- *value*, where *value* contains a string, then such string is contained in an element of type *StringExp* when the transformation is treated as a model

(cf. Figure 9). The specific text of the string is stored in the *stringSymbol* attribute of the *StringExp*. For this reason, the rule *CompleteAddInPatternElement* matches, in line 12, those elements of type *StringExp* whose *stringSymbol* attribute contains the text *Complete_IPE*. Consequently, it will match the *value* of the binding *name <- 'Complete_IPE'* defined in line 23 of Listing 2.

The purpose of the rule we are describing is to modify the *StringExp* element that has matched the rule by giving a different value to its *stringSymbol* attribute, what is done from line 16. A local variable defined in the *using* block (line 13), *classes*, is used in the right-hand side of the rule. In contains the set of classes of *IN_MM* that are not abstract. Please note as well that the helper *random()* defined in line 7 is also needed. It is specifically defined for the EMFTVM compiler of ATL [21], [22] and returns a random number between 0 and 1 whenever it is called. Consequently, the value assigned to *stringSymbol* is the name of one of the classes of *IN_MM* randomly chosen.

B. Discussion

As explained, the approach we have presented for defining HOTs with mutations comprises two steps. In the first place, HOTs with generic mutations are defined, and then, they are made specific by means of a second-order HOT. We have exemplified this process with the *AddInPatternElement* HOT with a generic mutation, which is then made specific by assigning a specific class to the *InPatternElement* that is created. However, there may be generic mutations that do not need to be made specific for a concrete transformation, i.e., they can be directly applied to any ATL transformation. An example is the *DeleteOutPatternElement* transformation shown in Listing 3. Since its purpose is to delete the last *OutPatternElement* of the last rule of the transformation, such deletion does not require any information about, for instance, a specific class of the input or output metamodels of the transformation to be mutated. However, if the user wants to remove an element in a specific rule of a specific transformation, then the *DeleteOutPatternElement* transformation should be made specific for the concrete scenario.

Another remark we want to make has to do with the behavior of the in-place mode of ATL, namely *refining mode*. As explained in [23], such behavior is not purely in-place. In a purely in-place mode, when a transformation rule is applied in a model, the output model produced by the rule is the input for matching the remaining rules (including the rule that has just been applied, if another match is found). However, in the refining mode of ATL, the model performing the matches with the rules is always the model that is given as input, not the evolved one. For this reason, if we try to execute the *DeleteOutPatternElement* transformation shown in Listng 3, we will realize that only the first rule is applied. This is because no matching with the original input model is performed in the second rule. For this reason, the two rules of the transformation have to be actually split in two transformations, and apply one after the other, so that the input model for the latter is the output model obtained by the former.

Our final comment has to do with the use of this approach for a final user interested in performing mutations to ATL transformations. In its current state, our proof-of-concept prototype (cf. Section V) with the artifacts shown in Figure 8(b) is completely automatic. This means that, once HOTs with generic mutations as well as the second-order HOT are defined, the prototype runs and mutates an ATL transformation with the mutations specified. However, it is also possible to develop a user-driven mutation approach. In this sense, users could specify which of the mutations they want to apply, so that different mutations and re-combinations of them are possible. This consists of turning our approach into a parametric approach. In fact, we envision the definition of a *Third-Order HOT* (cf. Figure 8(a)) that aims to generate the rules for the *HOTs with Generic Mutations* and the *Second-Order HOT*. In this way, the user could specify parameters, *User Options*, which could be defined in the form of a model according to the mutation possibilities. Two types of parameters can be differentiated. On the one hand, those related to which mutation operations the user wants to apply; they do not need any information about a specific transformation to be mutated. On the other hand, parameters related to features of specific mutations, such as in which rule an *InPatternElement* should be added; these may need information about a specific transformation. To show the feasibility of this approach, in Listing 5 we show a transformation to create a generic *DeteleOutPatternElement* rule that deletes an *OutPatternElement* of a rule—note that this transformation has been defined as an in-place transformation where the input would be a transformation with an empty module. It is not specified which *OutPatternElement* of which rule will be removed by the rule resulting from this transformation. Those parameters could be given by the *User Options*, or could be randomly chosen. The effect of parameterizing the rule that is produced by the transformation shown on Listing 5 would be the addition of a *Filter* in the produced rule. This *Filter* would restrict the concrete *OutPatternElement* that is to be deleted. Indeed, since transformations can be treated as models (cf. Section II-B), the *Filter* could be generated automatically by the *ThirdOrderHOT* transformation taken as input the user options.

Listing 5. Third-Order HOT.

```
1  -- @atlcompiler emftvm
2
3  module ThirdOrderHOT;
4  create OUT : ATL refining IN : ATL;
5
6  rule CreateRuleDeleteOutPatternElement{
7   from s : ATL!Module
8   to
9    t : ATL!Module(elements <- Sequence{rule1}),
10   rule1 : ATL!MatchedRule(
11     name <- 'DeleteOutPatternElement',
12     inPattern <- ip,
13     outPattern <- op ),
14   ip : ATL!InPattern(elements <- Sequence{ipe}),
15   op : ATL!OutPattern(elements <- Sequence{}),
16   ipe : ATL!InPatternElement(
17     varName <- 'ope',
18     type <- ome),
19   ome : ATL!OclModelElement(
20     name <- 'SimpleOutPatternElement',
21     model <- ATL!OclModel.allInstances() ->
22       select(m | m.name='ATL') -> first())
23 }
```

C. Cost-Effective Execution

As we have shown in this section, and as schematized in Figure 8, the goal of our approach is to automate the generation of mutants for ATL model transformations. A mutated transformation can be seen as an evolution of the original

transformation. Consequently, output models produced by the transformation need to co-evolve. However, the re-execution of the mutated transformations in order to obtain the evolved models induces an unnecessary overhead, particularly when computation-intensive transformations are marginally revised. To tackle the challenge of co-evolving output models with mutations in transformations, we have proposed to infer in-place patch transformations from evolved out-place transformations for existing output models [20]. Such approach is highly complementary to the one we have described in this paper, and can be used for executing the obtained mutated transformations for obtaining the new output models.

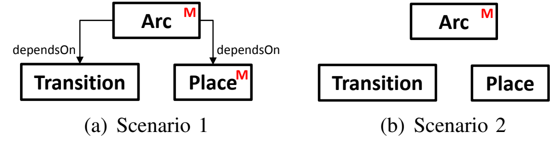
Our approach considers the original transformation and the mutated one, and produces a so-called *diff model* [24] that describes the differences between the two versions of the transformation. Subsequently, a HOT takes this diff model and the two versions of the transformation as input and produces a new transformation called *Patch Transformation*. It is an in-place transformation that defines the transformation rules required to co-evolve the output model according to the mutations in the transformation that has been executed to produce the model. It takes as input the output model produced from the original transformation and makes it co-evolve. The input model of the original transformation may also be an input for the patch transformation because it may need to be partially queried. The co-evolved output model produced is equal compared to the one gained from entirely re-executing the mutated transformation.

V. Prototype and Experiments

We have implemented a first proof-of-concept prototype with the implementation of the HOTs in the ATL language that we have shown in Section IV. The transformations are defined and executed using the EMFTVM virtual machine [21], [22], which offers advance features such as the possibility to call Java libraries. Besides, we have orchestrated the process shown in Figure 8 with a Java program that takes as input several HOTs, the transformation to be mutated and its input and output metamodels, and generates the mutated transformation[2]. The orchestration also realizes several extractions and injections of model transformations (cf. Section II-B). The reason is that if a transformation is to be the input of a HOT, it needs to be represented as a model. However, it the transformation is going to be executed, it needs to be represented in a file with the ATL textual syntax.

Listing 6 displays the mutated transformation obtained when applying the chain of transformations with the two mutations defined in Section IV. Furthermore, thanks to the program that computes the dependencies among rules described in Section III-C, the interested user can have a look at the dependencies graph. This can be useful for several reasons. For instance, we can see if the behavior of a rule, even though it has not been mutated, can influence the obtained output model because it depends on a rule that has actually been mutated. In our example, for instance, we can see that rule *Arc* has been mutated, and it has also been mutated a rule on which it depends (Figure 10(a)). It can also be useful to check if the mutations performed in the transformation have broken any of the rules dependencies. For instance, if we have a mutation where the *Bindings* in lines 36 and 37 of Listing 1 are removed, then rule *Arc* does not depend on the other two rules anymore (Figure 10(b)). Please note that the dependencies graphs obtained for this example are quite trivial. However, they are more interesting and useful when dealing with large transformations.

[2] This implementation is available on http://atenea.lcc.uma.es/index.php/Main_Page/Resources/Mutations



(a) Scenario 1 (b) Scenario 2

Fig. 10. Dependencies among rules.

Listing 6. *PetriNet2PNML* ATL Trans Mutated.
```
module PetriNet2PNML;
create OUT : PNML from IN : PetriNet;

rule Place {
  from
    e : PetriNet!Place,
    a : PetriNet!Place --InPatternElement Added
  to --...
}

rule Transition --...

rule Arc {
  from --...
  to
    n : PNML!Arc (
        name <- name,
        location <- e.location,
        id <- e.name,
        source <- e."from",
        target <- e."to"),
    name : PNML!Name --Binding deleted
    --OutPatternElement deleted
}
```

The program that obtains the dependencies among rules as explained in Section III-C is implemented as a HOT that takes as input the ATL transformation plus its input and output models, and produces as output a model conforming to the metamodel shown in Figure 7.

As for the effects that the mutated transformation has on the generated output models, we have used as test suite the input model available in the ATL Zoo for the *PetriNet2PNML* transformation. First of all, let us recall the mutation operators we have applied: we have added an *InPatternElement* to the rule *Place* and have deleted a *Binding* and an *OutPatternElement* from rule *Arc* (cf. Listing 6). In total, the mutant has 2 out of 4 transformation rules mutated (for simplification purposes, we have only shown 3 out of 4 rules of the *PetriNet2PNML* transformation in Listings 1 and 6). This means that it is very likely that an output model produced by the mutant is different from an output model produced from the same input model by the original transformation. In fact, by definition, a petri net contains places (although, according to the metamodel shown in Figure 1, we can have an object of type *PetriNet* without any place). For this reason, the mutated transformation will likely produce different output models than those obtained by the original transformation for all input models.

With the input model obtained from the ATL Zoo, which

consists of a petri net with 6 places, 6 places are produced in the PNML representation by the original transformation, what is done by rule *Place*. However, the mutant produces 36 places in the generated model. This is produced by the mutation of rule *Place* with the mutation operator *AddInPatternElement* explained in Section IV. The reason why so many places are generated by this mutation was explained in Section III-B and depicted in Figure 5. The mutant also has applied the mutation *DeleteOutPatternElement* shown in Listing 3. In the output produced by the original transformation, every place created has an object of type *Name*, which in turn contains an object of type *Label*. However, in the output produced by the mutant, the objects of type *Name* do not contain any object of type *Label*, as a result of the second mutation.

The mutated output models produced by the mutants can consequently be killed for any input model (considering that a petri net must contain at least one place). As future work, we plan to, first of all, implement more mutation operators. Then, we also plan to evaluate the mutants produced in different transformation scenarios and study how many mutants can be killed. For this, we will consider the application of single mutation operators in order to produce mutants, as well as the combination of several mutation operators for producing a mutant. We also want to apply the approach presented in our previous work [20] in order to optimize the time to produce mutants and to execute tests.

VI. RELATED WORK

As mutation has shown to be useful in assessing the adequacy of test suits, not surprisingly it is also applied in the context of testing model transformations to generate test data in terms of input models [11], [12] and mutants of model transformations [13]–[15]. Considering the latter case, these approaches produce generated mutants to be close to faults that may occur in a model transformation. Thereby, these approaches contribute new operators to the existing set of mutation operators which are already available for a multitude of languages [10].

Mottu et al. [13] propose generic mutation operators in the sense that they are independent of a particular transformation language. The mutation operators are classified into four main phases that a transformation passes through during execution: navigation, filtering, output model creation, and input model modification, where the latter phase is only passed through if the model transformation refines the input model. While the authors believe that the operators identified are meaningful since a large part of the errors in a transformation are due to the manupulation of complex models regardless of the concrete implementation language, we have preferred to focus on ATL. One reason is that by focusing on a specific language, we can specify and implement specific mutation operators for such language. Another reason for choosing ATL is that it is one of the most prominent hybrid transformation languages currently used in academia and industry. Fraternali and Tisi [14] also focus on ATL and demonstrate in their work how a generic mutation operator can be implemented for ATL by applying a HOT [19] that is also implemented in ATL. They selected the *Collection Filtering Change with Deletion (CFCD)* operator [13] and adopted it to ATL by mutating the *Filter*, which is used in ATL to constrain the possible matchings between elements of the input model and the output model. In the same spirit, we have applied HOTs to allow ATL model transformations to be mutated automatically by dedicated operators. However, we applied a comprehensive approach in identifying mutation operators for ATL by analyzing its metamodel according to how the application of the main concepts can be varied from a developer's perspective.

Closely related to our approach is the work of Khan and Hassine [15], as they also propose mutation operators specific to ATL and discuss the consequences when executing a mutated model transformation. However, we identified a more extensive set of mutation operators compared to their set, mainly because of our systematic analysis of ATL's metamodel. To efficiently produce a useful set of mutated model transformations, automation plays a key role, as also pointed out in the work of Khan and Hassine [15]. They manually applied some of their mutation operators in the context of a case study and left the development of a tool to automate this process for future work. In contrast, we set our focus on the development of a framework that is dedicated to automatically mutate ATL model transformations based on an extensive set of effective operators. Our framework also provides means for efficiently executing mutated transformations by exploiting incremental transformation execution instead of forcing a complete re-execution of the transformation. Thereby, only the transformation rules affected by the mutation are executed, which generally results in an improved runtime performance of the mutated transformations, as shown in our previous work [20].

Since in our approach, centered on the use of higher-order transformations, ATL transformations are treated as models when they are mutated, we consider that it is also worth mentioning some related work in the field of model-based mutation. For instance, Fabbri et al. [25] focus on the application of mutation testing in finite state machines (FSMs). The mutation operators they define are based on error classes and on heuristics that the authors have devised about typical errors made by designers during the creation of FSMs. Aichernig et al. [26] also deal with the model-based mutation testing for state machines, in this case those defined in UML. The authors propose to apply mutation operators both on the level of the specification to insert faults as well as to generate the test cases that will reveal the faults inserted. Henard et al. [27] focus on the ability of test suites to detect errors in software product lines, and propose two mutation operators to derive the mutants from an original feature model. Finally, Papadakis et al. [28] use models to represent the input interactions and apply mutation analysis on these models to select program test cases. Therefore, several approaches already address the mutation of models of different types. As mentioned, in our case we focus on models conforming to the ATL metamodel, i.e. ATL model transformations expressed as models, and use precisely models of the same type (HOTs defined in ATL, which in turn can also be seen as models conforming to the ATL metamodel) to mutate them. We currently focus only on the generation of mutants for the model transformations. However, for future work, we also want to focus on the automatic generation of test cases, for which we also plan to apply mutations. In this case, the models to mutate would then conform to different metamodels, so we would apply similar techniques as presented in this paper in order to define systematically

mutation operators and to generate mutants for these models through model transformations.

VII. Conclusion and Future Work

In this paper we have presented a novel approach to produce mutants for ATL transformations. To this end, we have identified an extensive set of mutation operators dedicated to ATL by a systematic analysis of its metamodel and described the effect they produce in the output model. We have automated the generation of mutants by realizing a framework that exploits the concept of HOTs, and we have integrated into our framework means to reduce the computational costs of executing model transformation mutants.

We have demonstrated the feasibility of our approach with a proof-of-concept prototype, with which we are able to mutate any ATL model transformation following the approach described in Figure 8(b). We have several ideas to realize next. We want to investigate the tendency for individual operators to produce either redundant or equivalent mutants. Other than the mutation operators identified, we plan to study more mutations in ATL by considering the imperative part of the language and mutations within OCL expressions. In particular, we want to study the effectiveness of the mutation operators, and specially to identify different mutation operators for *Filters*, which are defined with OCL expressions, as well as their consequences. Furthermore, we want to study the realization of the approach described in Figure 8(a), so that the selection of mutations to perform can be driven by the user, who could specify several options in the form of parameters not only for model transformations but also for languages and models defined with metamodels in general.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work is funded by the European Commission under ICT Policy Support Programme, grant no. 317859 and by Spanish Project TIN2011-23795.

References

[1] S. Sendall and W. Kozaczynski, “Model transformation: The heart and soul of model-driven software development,” *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[2] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan&Claypool, 2012.

[3] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu, “Barriers to Systematic Model Transformation Testing,” *CACM*, vol. 53, no. 6, pp. 139–143, 2010.

[4] E. Guerra, J. Lara, D. S. Kolovos, R. F. Paige, and O. M. Santos, “Engineering Model Transformations with transML,” *Softw. Syst. Model.*, vol. 12, no. 3, pp. 555–577, 2013.

[5] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, “Uncovering errors in ATL model transformations using static analysis and constraint solving,” in *Proc. of ISSRE’14*. IEEE, 2014.

[6] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, “Static Fault Localization in Model Transformations,” *IEEE Trans. Soft. Eng.*, 2015, accepted for publication.

[7] M. Amrani, L. Lucio, G. M. K. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy, “A tridimensional approach for studying the formal verification of model transformations,” in *Proc. of ICST*. IEEE, 2012, pp. 921–928.

[8] M. Gogolla and A. Vallecillo, “*Tract*able Model Transformation Testing,” in *ECMFA*, ser. LNCS, vol. 6698. Springer, 2011, pp. 221–235.

[9] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, “Formal specification and testing of model transformations,” in *Proc. of SFM*, ser. LNCS, vol. 7320. Springer, 2012, pp. 399–437.

[10] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[11] F. Fleurey, J. Steel, and B. Baudry, “Validation in Model-Driven Engineering: Testing Model transformations,” in *SIVOES-MoDeVA @ ISSRE*, 2004, pp. 29–40.

[12] V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser, “Towards an Automation of the Mutation Analysis Dedicated to Model Transformation,” *Software Testing, Verification and Reliability*, 2014.

[13] J.-M. Mottu, B. Baudry, and Y. Le Traon, “Mutation Analysis Testing for Model Transformations,” in *ECMDA-FA*, ser. LNCS, vol. 4066. Springer, 2006, pp. 376–390.

[14] P. Fraternali and M. Tisi, “Mutation Analysis for Model Transformations in ATL,” in *MtATL @ LSM*, 2009, pp. 145–149.

[15] Y. Khan and J. Hassine, “Mutation Operators for the Atlas Transformation Language,” in *ICSTW*, 2013, pp. 43–52.

[16] F. Jouault, “Loosely Coupled Traceability for ATL,” in *Workshop Proceedings of ECMDA*, 2005.

[17] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *SCP*, vol. 72, no. 1-2, pp. 31–39, 2008.

[18] A. Simão, J. C. Maldonado, and R. da Silva Bigonha, “A Transformational Language for Mutant Description,” *Comput. Lang. Syst. Struct.*, vol. 35, no. 3, pp. 322–339, 2009.

[19] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, “On the Use of Higher-Order Model Transformations,” in *ECMDA-FA*, ser. LNCS. Springer, 2009, vol. 5562, pp. 18–33.

[20] A. Bergmayr, J. Troya, and M. Wimmer, “From Out-place Transformation Evolution to In-place Model Patching,” in *ASE*. ACM, 2014, pp. 647–652.

[21] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, “Towards a General Composition Semantics for Rule-Based Model Transformation,” in *MODELS*, ser. LNCS, vol. 6981. Springer, 2011, pp. 623–637.

[22] D. Wagelaar and F. Jouault, “ATL/EMFTVM,” 2014, https://wiki.eclipse.org/ATL/EMFTVM.

[23] J. Troya and A. Vallecillo, “A Rewriting Logic Semantics for ATL,” *Journal of Object Technology*, vol. 10, pp. 5:1–29, 2011.

[24] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, “Different models for model matching: An analysis of approaches to support model differencing,” in *CVSM @ ICSE*, 2009, pp. 1–6.

[25] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro, “Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing,” in *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC’99)*, 1999, p. 96.

[26] B. K. Aichernig, H. Brandl, E. Jbstl, W. Krenn, R. Schlick, and S. Tiran, “Killing strategies for model-based mutation testing,” *Software Testing, Verification and Reliability*, pp. n/a–n/a, Feb. 2014.

[27] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, “Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing,” in *Workshops Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 188–197.

[28] M. Papadakis, C. Henard, and Y. L. Traon, “Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing,” in *IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST*. IEEE, 2014, pp. 1–10.