From Out-Place Transformation Evolution to In-Place Model Patching

Alexander Bergmayr
Vienna University of Technology, Austria
bergmayr@big.tuwien.ac.at

Javier Troya
Vienna University of Technology, Austria
troya@big.tuwien.ac.at

Manuel Wimmer
Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

ABSTRACT

Model transformation is a key technique to automate software engineering tasks. Like any other software, transformations are not resilient to change. As changes to transformations can invalidate previously produced models, these changes need to be reflected also on existing models. Currently, revised out-place transformations are re-executed entirely to achieve this co-evolution task. However, this induces an unnecessary overhead, particularly when computation-intensive transformations are only marginally revised, and if existing models undergone manual updates prior the re-execution, these updates get discarded in the newly produced models.

To overcome this co-evolution challenge, our idea is to infer from evolved out-place transformations so-called *patch transformations* that facilitate the propagation of changes to existing models by re-executing solely the affected parts based on an in-place execution strategy. Thereby, existing models are only updated instead of newly produced. In this paper, we present the conceptual foundations of our approach and report on its evaluation in a real-world case study.

1. INTRODUCTION

Model transformation is a key technique to automate several kinds of software engineering tasks [3, 22]. Transformations are employed in reverse/forward engineering processes and for exchanging models between different tools (cf. [1] for more use cases). These kinds of transformations are often implemented as *out-place transformations* [16], i.e., the output models are built from scratch by executing the transformation on the input models.

Problem. Like any other software, transformations are not resilient to change [20, 24, 26]. As changes to transformations can easily invalidate previously produced models, these changes need to be propagated to existing models. In the rest of this paper we refer to this situation as *transformation/output model co-evolution*, i.e., the transformation evolution implies an evolution, thus co-evolution, of the existing output models. Currently, revised out-place transformations have to be entirely re-executed to achieve this co-evolution task. However, this induces an unnecessary overhead, particularly when computation-intensive transformations are only marginally revised, and if existing models undergone manual updates prior the re-execution, these updates get discarded in the newly produced models. Furthermore, if models are referencing each other based on identifiers, re-creating the output models from scratch can break these inter-model references.

Contribution. To tackle the problem of transformation/output model co-evolution, we introduce an approach for deriving *in-place patch transformations* for already existing output models from *evolving out-place transformations*. The generated patch transformations take care of propagating the changes done for evolving the out-place transformation to the output models without re-creating them from scratch. Thereby, these patch transformations only update the existing models [16] where changes are necessary following an in-place execution strategy, i.e., they are *patching* the models to be conform with respect to the new transformation. The proposed approach fills the gap between current research focussing on *incremental transformations* [6, 8, 10, 13, 18, 19] for propagating changes in input models to already existing output models, and *metamodel/transformation co-evolution* for propagating changes in metamodels to transformations [7, 9, 15, 21]. In this paper we report on the conceptual foundations of our approach, an implementation for generating in-place patch transformations for out-place transformations defined in the ATLAS Transformation Language (ATL) [12], and report evaluation results gained in a real-world case study considering the translation of Java models to UML models.

Structure. In Section 2, we present the evolution dimensions of model transformations, summarize related approaches, and formulate the problem statement for this paper. We present our approach in Section 3 by discussing our conceptual framework and prototypical implementation based on the Eclipse environment. In Section 4 we evaluate our approach. Finally, we conclude in Section 5 with an outlook on future work.

2. PROBLEM STATEMENT

The background of this work is the model transformation pattern [5], which considers the systematic transformation of *input models* conforming to *input metamodels* into *output models* conforming to *output metamodels* by executing a transformation. For implementing transformations, several model transformation languages emerged in the last decade which provide certain characteristics. Most importantly, the language paradigm they are following may be classified in declarative, imperative, and hybrid. Furthermore, the execution possibilities of transformations is of major interest. While several languages allow for an *uni-directional* execution only, some languages allow to transform in both directions as well as to match and synchronize existing input and output models. In this paper, we are focussing on uni-directional transformation languages, in particular on ATL which is one of the most promi-

nent languages currently used in academia and industry.

To establish the basis for our investigations, we discuss in the following the evolution dimensions one is confronted with when working in the field of model transformation engineering. Each evolution dimension we discuss has an *initial evolution step* and an associated *co-evolution step*. Recently, the following two evolution dimensions (cf. Figures 1(a) & 1(b)) have been investigated.

First, there are approaches that consider the evolution of metamodels used by transformations as illustrated in Figure 1(a). As the metamodels contribute important parts of the type systems for transformations, if they change, also the transformations are influenced and require co-evolution actions. This evolution dimension is often referred to as metamodel/transformation co-evolution and is supported by [7, 9, 15, 21]. The main goal of these approaches is to (semi-)automatically adapt the transformation to the new metamodel versions and preserve the transformation's behavior as much as possible.

Second, the evolution on model level has to be considered. If the input models evolve, the output models have to co-evolve as is shown in Figure 1(b). This could be straight-forwardly achieved by running the entire transformation in batch mode for the new version of the input model. Due to several reasons such as reducing computation time, e.g., in case of minor changes on large models or preserving manual updates in the output models, an incremental execution of the transformation is favoured, i.e., only the changes in the input models should be propagated by the transformation to the output models. There are several approaches that allow for incremental execution of model transformations with respect to changes in the input models, cf. [6] for a survey and [8, 10, 13, 18, 19] for specific approaches. The main idea behind incremental transformation is that the runtime complexity of a transformation is no longer proportional to the size of the input models but to the size of the changes performed on the input models.

Besides these two evolution dimensions which received already attention in research, there is another scenario (cf. Figure 1(c)) that may be seen as the intersection of the previous two and, to the best of our knowledge, has not been tackled, yet. If we assume we already have several transformation runs for a transformation, i.e., there are already several output models produced from input models, and this transformation evolves, e.g., due to fixing a bug in the transformation, all transformation runs have to be reproduced to generate valid output models, i.e., to reflect the update in the transformation in the transformation results. Clearly, here we would like to gain the same benefits as incremental transformations provide for propagating changes in the input models to the output models. But instead, in this third scenario, we have to propagate the changes in the transformation to the existing output models. This leads to the following problem statement of this paper: *How can we achieve incremental transformation executions which propagate changes in the transformations to the existing output models?*

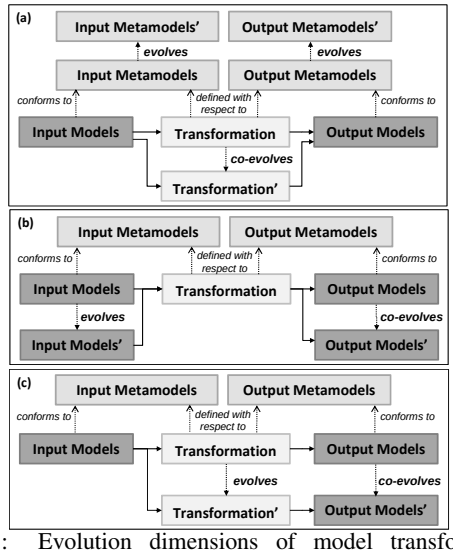

Figure 1: Evolution dimensions of model transformations: (a) metamodel/transformation co-evolution, (b) input/output co-evolution, (c) transformation/ouput co-evolution.

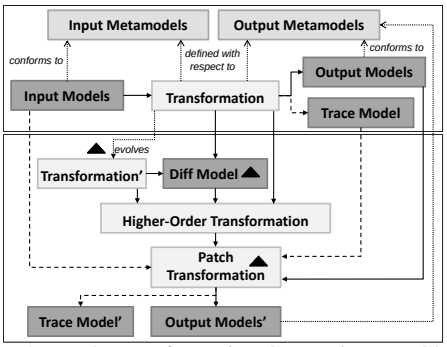

Figure 2: Patch Transformation Generation at a Glance.

3. PATCH TRANSFORMATIONS

In this section, we present how we tackle the challenge of propagating changes in transformations to previously produced output models. The general idea behind our approach is to reformulate the changes done in the out-place transformations as in-place transformations capable of propagating these changes to the output models.

3.1 Approach at a glance

Figure 2 gives an overview of our approach. The upper part considers the model transformation pattern as discussed before. In the figure, filled lines represent inputs and outputs, thick dashed lines represent optional inputs and outputs, and thin dashed lines represent conceptual and conformance relationships as well as evolution. Regarding the upper part, a model transformation receives a set of models as input and produce a set of models as output. Optionally, it can also produce a special kind of output model automatically, namely a *trace model*, that captures the relationships between the elements in the input and output models by defining trace links. Each trace link corresponds to the execution of a transformation rule as we will see later in more detail.

The lower part begins with the evolution of the transformation and explains the main steps of our approach. Here, evolution means that at least one change has occurred in the original transformation. The set of change types that we consider in this paper is described in Section 3.3. With the original transformation and the evolved one, we produce a so-called *diff model* [14], which describes the differences between both transformation versions. Subsequently, a *Higher-Order Transformation* (HOT)[1] [23] takes this model as well as the original and evolved transformations as input, and produces a new transformation, namely a *Patch Transformation* (PT). The PT obtained after the execution of the HOT considers only those parts that need to be evolved in the output models produced by the original transformation.

The obtained PT can have up to three (sets of) input models. In the first place, the set of output models produced from the original transformation have to be an input, since these will be evolved

[1]According to [23], "*a HOT is a model transformation such that its input and/or output models are themselves transformation models*".

according to the changes indicated in the PT. Other than these, the trace model and set of input models are optional inputs for the PT. Evidently, the more information we have as input, the more accurate the evolved output models can be. In this work, we consider that we have the three inputs shown in the figure and that the model produced by the patch transformation is the same as the one we would get from executing the complete evolved transformation. The PT also produces a patch for the trace model, if it receives one as input. In this case, the produced trace model is the same as if the evolved transformation had been executed from scratch.

3.2 Transformation Language Elements

We have chosen the ATL language as a proof-of-concept for our approach. It is a hybrid model transformation language containing a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. As we will see later, this is an important property of ATL to facilitate patch transformations. In this paper, we are focussing on ATL's declarative part.

An excerpt of the ATL metamodel for its declarative part is shown in Figure 3(a). A *Transformation* is composed of one or more declarative *MatchedRule*s, and has one or more *Model*s as input and output. *MatchedRule*s contain one *InPattern* and one *OutPattern*. The former is a query on the input model, and gathers the set of *InPatternElement*s that represent the input model elements of the rule. It can also contain a *Filter*, which is composed by conditions that the *InPatternElement*s have to satisfy to apply the rule. *Filter*s are specified by means of Object Constraint Language (OCL) expressions. The *OutPattern*, in turn, describes the creation of elements in the target model. Such elements are of type *OutPatternElement*s. Each *OutPatternElement* is composed of a set of *Binding*s, whose values are expressed and computed by OCL expressions, used to initialize the features of output model elements.

As for the semantics of ATL, the order in which the rules are written does not affect the computation of output models, due to a two-phased process. In the first phase, the matching conditions of the rules, i.e., *InPattern*s, are tested and the ATL execution engine simply allocates the set of output model elements that correspond to the *OutPattern*s declared in the rules. In the second phase, the output model elements are initialized, i.e., their feature values are set by the bindings.

Listing 1 depicts an excerpt of the *Java2UML* transformation, which is currently developed and continuously revised in the ARTIST project [2]. Currently, the transformation consists of 29 rules. It

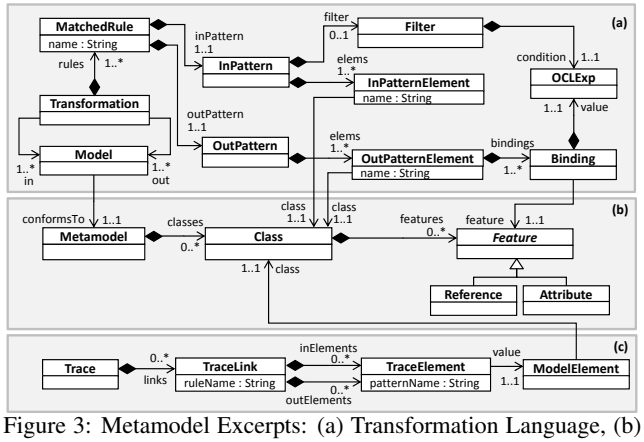

Figure 3: Metamodel Excerpts: (a) Transformation Language, (b) Metamodeling Language, and (c) Trace Language.

is defined according to the Java metamodel (*JMM*) provided by MoDisco [4] and the UML metamodel (*UMLMM*) that comes with the Eclipse modeling distribution. Basically, Java packages and class declarations are transformed to their UML correspondences. For the purpose of demonstrating a running example we assume that only class declarations contained by certain packages, i.e., *domain* and *web*, are considered. This behavior is achieved by the defined *Filter* of the *InPattern* in the second *MatchedRule*. The *Binding*s of the two rules ensure that the names of the packages and classes as well as the references in-between are pushed from the Java models to the produced UML models. For demonstration purposes, we assume a revision of the shown transformation, which refers to the *Filter* condition of the second rule. Thereby, class declarations contained by the *service* package instead of the *web* package are expected in the produced UML model.

Listing 1: *Java2UML* ATL Transformation Evolution

```
rule Pack2Pack{
 from s:JMM!Package
 to t:UMLMM!Package(
    name <- s.name,
    packagedElement <- s.ownedElements )}

rule Class2Class{
 from s:JMM!ClassDeclaration(
      Set{'domain','web'}->includes(s.package.name)
      Set{'domain','service'}->includes(s.package.name))
 to t:UMLMM!Class( name <- s.name )}
```

Let us comment here on a feature of transformation languages that we call inter-rule dependencies. In our transformation example, the *return type* of the value expression (*s.ownedElements*) of binding *packagedElement* is *Sequence(JMM!AbstractTypeElement)*, thus it may also contain classes (*JMM!ClassDeclaration* is a subtype of *JMM!AbstractTypeElement*). For this reason, when this binding is computed, *packagedElement* will reference, among others, those elements created in the second rule[2]. It is important that we deal *explicitly* with these dependencies when creating patch transformations, since, focusing on our example, a modification in the second rule may cause the re-computation of binding *packagedElement* of the first one. To deal with these situations, we perform a static analysis on the revised transformation. It consists of using a HOT to first determine the return types of value expressions of bindings and second calculate the dependencies between bindings and rules. As we explain in Section 3.3, we use these explicit dependencies when generating the patch transformations.

Finally, we also use in our approach an explicit trace metamodel (cf. Figure 3(c)). In fact, we can automatically obtain a trace model from a transformation run, e.g., by using Jouault's *TraceAdder* [11]. A *Trace* is composed of *TraceLink*s. A *TraceLink* keeps the name of the producer *MatchedRule* and contains *TraceElement*s. These elements contain the *name* of the corresponding *InPatternElement* or *OutPatternElement* and reference to the input or output model elements which have been queried or generated. To sum up, by looking at the trace model, we can see all rule executions as well as which input model elements contributed to the generation of specific output model elements. An example trace model excerpt for a possible execution of the Java2UML transformation is shown in Figure 4.

3.3 Change Types and Patch Requirements

The different kinds of changes that we have considered for the evolution of a transformation as well as its co-changes for the al-

[2]ATL performs a transparent lookup of output elements for given input elements when executing bindings, e.g., for the queried JMM elements the corresponding UML elements are retrieved.

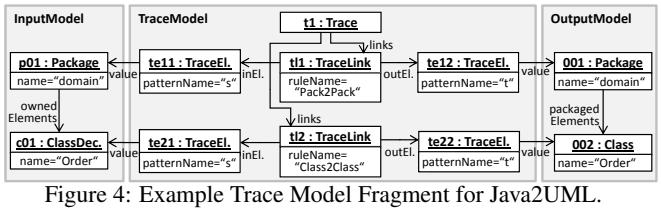

Figure 4: Example Trace Model Fragment for Java2UML.

ready existing output models are shown in Table 1. We aim for completeness of our approach by using a systematic approach where we consider the addition and deletion of instances for any metaclass in the Transformation Metamodel (Figure 3(a)), as well as update of their features.

The addition or deletion of a *MatchedRule* imply to add or delete the elements that such rule creates, respectively, while the change in its name simply implies to propagate such change to the trace model, to keep it properly updated. As for *InPatternElement*s, their addition/deletion in the *InPattern* means that the elements from the input model with which the rule matches change completely. For instance, if we had only one *InPatternElement* and we add another one, the match is realized now with the cartesian product of both elements. Contrarily, if we go from two *InPatternElement*s to only one, the rule has to create now much less elements. Furthermore, the addition/deletion of an *InPatternElement* in the *InPattern* implies that the *OutPattern* of the rule has also likely changed by adding/removing the corresponding variable referring to the new/old *InPatternElement* in one or several *Binding*s of *OutPatternElement*s. For this reason, we have decided to delete the changes produced by the rule and re-execute it again. Something similar happens when the *class* feature of an *InPatternElement* changes. In this case, it is like considering that a deletion and an addition have taken place. As for the modification of its *name* feature, we simply need to propagate such change in the trace model. Regarding *Filter*s, we consider that the effect of adding, deleting, or modifying them has to be the same. This is because, even if there is no *Filter*, it is like considering that there is one whose *condition* is simply *true*. In the same way, if a *Filter* is removed, it is the same as changing it to *true*. Consequently, we consider the three cases as if the *Filter* is modified. Thereby, its effect is to add those elements in the output model created from those elements in the input model that now satisfy the *Filter* and execute the corresponding *Binding*s, as well as to delete those elements in the output model that were created from input elements that used to satisfy the *Filter* but not anymore.

The modification of the two last concepts have simpler and more straight-forward effects. In the first place, adding or removing an *OutPatternElement* simply imply the addition (and creation of corresponding *Binding*s) of the element in the output model. If the *class* feature changes, we consider the addition and deletion of the *OutPatternElement*, and the modification of the *name* feature is simply propagated to the trace model. As for *Binding*s, if they are added or removed, then the effect is to compute their value or delete the value that was computed, respectively. If the *value* of an existing *Binding* changes, then it has to be recomputed and reassigned. If it is the *feature* feature what has changed, then we consider it as deletion and addition of the *Binding*.

Apart from the co-changes described above, we also have to take into account the dependencies between bindings and rules as explained in Section 3.2. Thereby, every time there is any change in *MatchedRule*s (except for modification of *name*), *InPatternElement*s (except for modification of *name*), *Filter*s and *OutPatternElement*s (except for modification of *name*), we need to check if such change is involved in an explicit dependency. If it is, this will

Table 1: Change Types

Concept	Change Type	Co-Change (Intended Effect of Patch Transformation)
MatchedRule	Addition	Execution of MatchedRule
	Deletion	Deletion of objects and features in output model generated by MatchedRule
	Modification (*name* feature)	Propagation of name change to trace model
InPatternEl	Addition Deletion	1. Deletion of objects and features in output model generated by MatchedRule 2. Execution of the MatchedRule as new rule
	Modification (*class* feature)	Considered as Deletion and Addition of InPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName*)
Filter	Addition Deletion Modification (*condition* feature)	1. Deletion of objects in output model created from objects in the input model that do not satisfy the Filter anymore 2. Creation of objects in output model from objects in the input model that now satisfy the Filter 3. Execution of Bindings
OutPatternEl	Addition	1. Creation of object in the output model 2. Execution of its Bindings
	Deletion	Deletion of previously created object in output model
	Modification (*class* feature)	Considered as Deletion and Addition of OutPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName*)
Binding	Addition	Execution of added Binding
	Deletion	Deletion of feature values
	Modification (*value* feature)	Re-execution of changed Binding
	Modification (*feature* feature)	Deletion and Addition of Binding

be reflected in the patch transformation, since the involved bindings have to be recomputed.

The way our differences are executed to produce the evolved output models follow the same semantics ATL applies for full execution (cf. Section 3.2). Thus, in a first phase, the output model elements are additionally created or deleted. Subsequently, in the second phase, the features of the output model elements are initialized where necessary, which implies the execution of the effected bindings.

Listing 2 presents the patch transformation inferred from the revision applied to the original transformation (see Listing 1). In the first rule, produced elements of existing models that do not satisfy the revised filter condition are deleted, whereas in the second rule, elements that have previously not satisfied the filter condition but satisfy the revised one are produced. In our example, these elements refer to UML classes of the original output model. The pertinent UML classes are computed prior the execution of the patch rules and provided by the respective sets, i.e., *rems* and *adds*. They are accomplished by executing the original and revised filter conditions against the original input model and calculating their differences. The third rule is dedicated to the re-execution of bindings affected by transformation revisions. Even though in our example the binding itself has not been explicitly revised, its re-execution is required to ensure that newly added UML classes are appropriately referenced by their containing UML packages. As such packages can already contain classes, the *union* operator needs to be applied to accomplish the expected result. In this respect, the pertinent Java packages are required as well. They are queried from the trace model in the *using* part of the patch transformation.

Listing 2: *Patch Transformation* in ATL

```
rule PatchFilterDelete {
 from s:UMLMM!Class(thisModule.rems->includes(s))}
```

```
rule PatchFilterAdd {
 from s:JMM!ClassDeclaration(thisModule.adds->
      includes(s))
 to t:UMLMM!Class( name <- s1.name )}

rule Pack2Pack {
 from
  ps:UMLMM!Package
  using {s:JMM!Package = thisModule.tls->any(e|e.
       outElements->exists(f|f.value = ps)).
       inElements->any(e|e.patternName = 't').
       value;}
 to
  t:UMLMM!Package(
   packagedElement <- ps.packagedElement->union(
        JMM!ClassDeclaration.allInstances() ->
        select(e|s.ownedElements->includes(e) and
        thisModule.adds->includes(e))))}
```

3.4 Implementation

We have implemented our approach as an experimental prototype in Eclipse as an open source project [17]. As a pre-step for computing the differences between ATL transformation versions, we inject the transformation code to a model representation by using the ATL injector component. By having the transformations in model representation, we employ EMF Compare[3] for computing the differences between them. To have a more concise diff model for our purposes, we provide some aggregation of diff elements as post-processing of the comparison in order to have a solid basis for producing the patch transformations. For generating the patch transformations from the diff models, Xtend[4] is used for directly generating ATL code which uses the refinement mode, i.e., the in-place mode, of ATL. Finally, for executing the patch transformations we use the EMF Transformation Virtual Machine (EMFTVM)[5] [25] due to some advanced features and support of true in-place execution instead of copying the input model entirely, as it is done in the standard ATL virtual machine.

4. EVALUATION

To evaluate our approach, we studied the *Java2UML* reverse-engineering case study that is developed and continuously evolved in the ARTIST project by means of model transformations. With this case study, we aim to answer the two research questions:

RQ1: *Are patch transformations equally effective as the revised transformations based on which they are inferred?*

RQ1: *How is the speed-up of executing patch transformations compared to re-executing the respective revised transformation?*

4.1 Case Study Setup

For the purpose of our case study, we selected a *Java2UML* transformation which has an extensive and well-documented evolution. As input models for this transformation in the case study of this paper, we selected a reference application of the ARTIST project, which is based on the Java Petstore[6], and a framework that is of high relevance in this respect: EclipseLink[7]. The main rational behind the reverse-engineering of the frameworks is to provide their annotations at the model level in terms of corresponding UML stereotypes (cf. UML-Profile-Store[8]). To generate the respective Java models for the reference application and EclipseLink, we employed MoDisco. To cover the presented change types and the core effects of patch transformations, we selected six different revisions of the *Java2UML* transformation that have been performed throughout its development. Based on these revisions, we inferred the corresponding patch transformations.

To answer *RQ1*, in a first step, we executed both the revised transformations and the inferred patch transformations. The produced output models of the transformations are the basis to investigate on the effectiveness of patch transformations compared to their corresponding revised transformations. Then, in a second step, we passed the respective pairs of output models to EMF Compare to automate the comparison task. Clearly, the diff model computed by EMF Compare needs to be empty to show that the produced output models are equal. It is important to note that the element identifiers can be different in the output models, as patch transformations preserve them while they are newly produced if revised out-place transformations are re-executed. To answer *RQ2*, we measured and compared the pertinent execution times of patch transformations and their respective revised transformations. For obtaining the measures, we executed the transformations in the Eclipse environment on commodity hardware: Intel Core i5-2520M CPU, 2.50 GHz, 8,00 GB RAM, 64 Bit OS. Thereby, a first impression of possible performance improvements by executing patch transformations instead of entirely re-executing revised transformation is given. All relevant artifacts of our case study are available at our project website [17].

4.2 Case Study Results

Considering the output models of the revised transformations and the patch transformations, their comparison shows that our approach produces effective results. In fact, the updates of the patch transformations to the output models reflect exactly the intended effects of the revisions performed to the original transformation. Clearly, as patch transformations only update the output models based on an in-place execution strategy, identifiers of existing elements and possible manual changes to elements that need not to be patched are preserved.

Turning the focus to the runtime efficiency of our approach, generally, our inferred patch transformations execute less rules compared to the revised transformations. The number of required rules of a patch transformation slightly varies depending on the considered change type. In our case study, one up to six rules were required to produce a patch transformation. Clearly, this number increases if certain rules need to be re-executed as a result of revising another rule (cf. inter-rule dependencies). Such dependencies lead to patch transformations covering not only revised rules but also rules that are affected by the performed revisions, e.g., bindings, as shown in Listing 1. Finally, the adaptation of the trace model requires also additional rules in the patch transformation, e.g., when matched rules are added. While the number of rules have certainly impact on the execution time of patch transformations, our results show that their need to traverse and query the traces of the original transformation produces an overhead compared to the revised transformations. Still, in our case study, for the majority of patch transformations a speed-up can be observed, as summarized in Table 2. In fact, only in one case, such a speed-up could not be achieved as the input and output models of the reference application are rather small and the inferred patch transformation for this case is more complex compared to other ones. However, the benefit of patch transformations to guarantee a non-invasive update to the output models is in our case study always given.

Threats to validity. In this paper, we set the focus on the *Java2UML*

[3] www.eclipse.org/emf/compare
[4] www.eclipse.org/xtend
[5] wiki.eclipse.org/ATL/EMFTVM
[6] www.oracle.com/technetwork/java
[7] www.eclipse.org/eclipselink
[8] code.google.com/a/eclipselabs.org/p/uml-profile-store

	Reference Appliction (> 1000 Elements)			EclipseLink (> 100.000 Elements)		
	Transformation		Speed-Up	Transformation		Speed-Up
Change Type	Revised	Patch		Revised	Patch	
MatchedRule Addition	0,076	0,067	1,134	6,698	4,766	1,405
MatchedRule Deletion	0,057	0,014	4,071	6,114	1,417	4,315
Filter Modification	0,066	0,079	0,835	5,854	4,987	1,174
OutPattern Element Addition	0,066	0,058	1,138	7,531	3,149	2,392
Binding Addition	0,063	0,010	6,300	6,687	0,104	64,298
Binding Deletion	0,064	0,009	7,111	6,882	0,058	118,655

Table 2: Performance Measures: Re-execution vs. Patch Execution

case study and apply patch transformations on small to large models that represent real-world applications and frameworks. Concerning internal validity, we need further exploration of different combinations of changes if they can be detected and propagated correctly and efficiently. Concerning external validity, we cannot claim any results outside of our performed case study concerning other transformation languages or transformations. We leave these considerations as subject to future work.

5. CONCLUSION AND FUTURE WORK

In this paper, we posed the problem of transformation/output model co-evolution and tackled this problem by reformulating changes on out-place transformations as in-place patch transformations for already existing output models. We demonstrated our approach for ATL and showed its benefits in a real-world case study. While the results already look promising, several lines of future work remain. First, we plan to tackle the problem of re-calculating trace models in cases where they are missing by transforming the out-place transformation in a match transformation. Second, we want to explore the extreme case where the input models are missing for existing output models. Here the question is how far we can keep the output models conform to the new transformation versions.

Acknowledgement

This work is co-funded by the European Commission under the ICT Policy Support Programme, grant no. 317859.

6. REFERENCES

[1] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Towards a model transformation intent catalog. In *Analysis of Model Transformations Workshop @ MODELS*, 2012.

[2] A. Bergmayr, H. Bruneliere, J. L. Cánovas Izquierdo, J. Gorroñogoitia, G. Kousiouris, D. Kyriazis, P. Langer, A. Menychtas, L. Orue-Echevarria Arrieta, C. Pezuela, and M. Wimmer. Migrating Legacy Software to the Cloud with ARTIST. In *CSMR*, 2013.

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

[4] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In *ASE*, 2010.

[5] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[6] J. Etzlstorfer, A. Kusel, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, and M. Wimmer. A Survey on Incremental Model Transformation Approaches. In *Models & Evolution Workshop @ MoDELS*, 2013.

[7] J. García, O. Díaz, and M. Azanza. Model Transformation Co-evolution: A Semi-automatic Approach. In *SLE*, 2012.

[8] D. Hearnden, M. Lawley, and K. Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In *MoDELS*, 2006.

[9] L. Iovino, A. Pierantonio, and I. Malavolta. On the Impact Significance of Metamodel Evolution in MDE. *JOT*, 11(3):3:1–33, 2012.

[10] S. Johann and A. Egyed. Instant and incremental transformation of models. In *ASE*, 2004.

[11] F. Jouault. Loosely Coupled Traceability for ATL. In *Workshop Proceedings of ECMDA*, 2005.

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *SCP*, 72(1-2):31–39, 2008.

[13] F. Jouault and M. Tisi. Towards Incremental Execution of ATL Transformations. In *ICMT*, 2010.

[14] 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 Workshop @ ICSE*, pages 1–6, 2009.

[15] T. Levendovszky, D. Balasubramanian, A. Narayanan, and G. Karsai. A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In *SLE*, 2009.

[16] T. Mens and P. V. Gorp. A taxonomy of model transformation. *ENTCS*, 152:125–142, 2006.

[17] Patch Transformations. `http://code.google.com/a/eclipselabs.org/p/patch-transformations`, 2014.

[18] I. Ráth, G. Bergmann, A. Ökrös, and D. Varró. Live Model Transformations Driven by Incremental Pattern Matching. In *ICMT*, 2008.

[19] A. Razavi and K. Kontogiannis. Partial Evaluation of Model Transformations. In *ICSE*, 2012.

[20] A. Rentschler, Q. Noorshams, L. Happe, and R. Reussner. Interactive Visual Analytics for Efficient Maintenance of Model Transformations. In *ICMT*, 2013.

[21] D. D. Ruscio, L. Iovino, and A. Pierantonio. A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations. In *ICMT*, 2013.

[22] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[23] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *ECMDA-FA*, 2009.

[24] M. van Amstel and M. G. J. van den Brand. Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In *ICMT*, 2011.

[25] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. Towards a General Composition Semantics for Rule-Based Model Transformation. In *MODELS*, 2011.

[26] M. Wimmer, G. Kappel, J. Schönböck, A. Kusel, W. Retschitzegger, and W. Schwinger. A Petri Net Based Debugging Environment for QVT Relations. In *ASE*, 2009.