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

Alexander Bergmayr, Javier Troya, and Manuel Wimmer
Vienna University of Technology, Austria
[lastname]@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 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 marginally revised, and if existing models have undergone 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 *patch transformations* that propagate changes to existing models by re-executing solely the affected parts based on an in-place execution strategy. Thereby, existing models are only updated by a *patch* instead of newly produced. In this paper, we present the conceptual foundation of our approach and report on its evaluation in a real-world case study.

1. INTRODUCTION

Model transformation is a key technique to automate software engineering tasks [6, 25]. Transformations enable reverse-engineering and forward-engineering scenarios and facilitate exchanging models between tools [2]. They are often implemented as *out-place transformations* [19], where the output models are built from scratch by executing the transformation on the input models.

Like any other software, transformations change over time [23, 27, 30]. As changes to transformations can invalidate previously produced models, these changes need to be propagated to existing models. We refer to this challenge as *transformation/output model co-evolution*, where changes to a transformation imply an evolution. Consequently, output models need to co-evolve. 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 marginally revised, and if existing models have undergone manual updates prior the re-execution, these updates get discarded in the newly produced models. Furthermore, if models reference each other based on identifiers, re-creating the output models from scratch can break these inter-model references.

To tackle the challenge of co-evolving output models with changes in transformations, we propose to infer *in-place patch transformations* from *evolved out-place transformations* for existing output models. A patch transformation enables propagating changes of an evolved out-place transformation to the pertinent output models without re-creating them from scratch. Hence, a patch transformation only updates existing models [19] according to an evolved transformation by an in-place execution strategy that enables the *patching* of these models. Our approach fills the gap between current research on *incremental transformations* [9, 11, 13, 16, 21, 22] for propagating changes in input models to already existing output models and *metamodel/transformation co-evolution* for propagating changes in metamodels to transformations [10, 12, 18, 24].

In Section 2, we present dimensions of model transformation evolution, summarize work related to our approach, and formulate the statement of the *transformation/output model co-evolution* problem. The conceptual foundation for our approach and the generation of in-place patch transformations for out-place transformations implemented in ATL [15] is discussed in Section 3. Finally, we report on evaluation results gained from a real-world case study for translating Java models to UML models in Section 4 before we conclude in Section 5 with an outlook on future work.

2. PROBLEM STATEMENT

The background of this work is the model transformation pattern [8]. It describes the systematic transformation of *input models* conforming to *input metamodels* into *output models* conforming to *output metamodels*. To implement transformations, several languages with different characteristics emerged in the last decade. Most importantly, their underlying paradigm can be classified in declarative, imperative, and hybrid. Furthermore, the execution possibilities of transformations is of major interest. While some languages enable *uni-directional* execution only, others are capable to transform in both directions and to match and synchronize existing input models and output models. In this paper, we set the focus on uni-directional languages and present our approach according to ATL, which is one of the most prominent hybrid languages currently used in academia and industry.

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

There are approaches that consider the evolution of metamodels based on which transformations are defined (cf. Figure 1(a)).

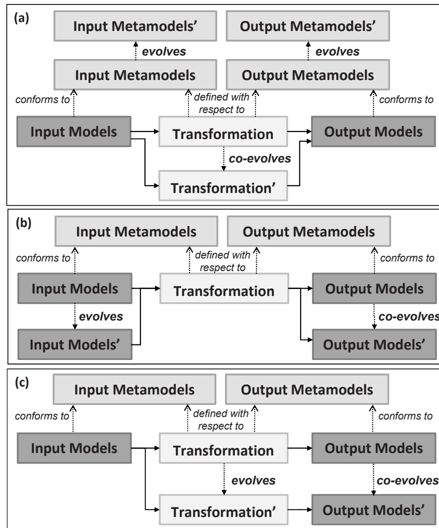

Figure 1: Dimensions of model transformation evolution: (a) metamodel/transformation co-evolution, (b) input/output co-evolution, (c) transformation/output co-evolution.

Metamodels contribute important parts to the type system of transformations. Consequently, if they change, the transformations are also influenced and may require co-evolution actions. This evolution dimension is often referred to as metamodel/transformation co-evolution and is supported by [10, 12, 18, 24]. The main goal of these approaches is to (semi-)automatically adapt transformations to new metamodel versions while preserving their behavior.

The evolution at model level poses another challenge. If input models evolve, the respective output models have to co-evolve (cf. Figure 1(b)). This could be straightforwardly achieved by executing the entire transformation in batch mode for new versions of input models. 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 preferred. Thereby, only changes in input models are propagated by the transformation to output models. There are several approaches that allow incremental execution of model transformations with respect to changes in input models; consider [9] for a survey and [11, 13, 16, 21, 22] for specific approaches. The main idea behind incremental transformations is that the runtime complexity of a transformation is no longer proportional to the size of input models but instead to the size of changes performed on them. A related approach refers to change-driven transformations [3] that react to changes in one model by propagating them to other models.

Besides these two evolution dimensions, which received already attention in research, we identified a third one (cf. Figure 1(c)) that can be considered as the intersection of the two discussed dimensions and, to the best of our knowledge, has not yet been tackled. Assuming that a transformation based on which output models have already been produced from input models evolves, e.g., due to fixing a bug in the transformation, all transformation executions have to be reproduced to turn existing output models into valid transformation results. Clearly, the same benefits incremental transformations provide for propagating changes in input models to output models are desirable for reflecting changes in transformations to previously produced output models. This leads to the following problem statement of our work: *How can we achieve incremental transformation execution that propagates changes in transformations to existing output models?*

3. PATCH TRANSFORMATIONS

We now discuss how the challenge of propagating changes in transformations to previously produced output models is tackled. The general idea of our approach is to reformulate changes done in out-place transformations as in-place transformations that are capable of propagating these changes to the respective 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 in Section 2. Filled lines represent inputs and outputs while thick dashed lines represent optional inputs and outputs. Thin dashed lines represent conceptual and conformance relationships, and indicate evolution. Regarding the upper part, a model transformation receives a set of models as input and produces a set of models as output. Optionally, it can automatically produce a special kind of output model, namely a *trace model*. It captures the relationships between elements of input models 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 describes the evolution of transformations and captures 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* [17] that describes the differences between the two transformation versions. Subsequently, a *Higher-Order Transformation* (HOT)[1] [1, 26, 28] takes this diff model and the two transformation versions as input and produces a new transformation called *Patch Transformation (PT)*. It defines the transformation rules required to co-evolve existing output models according to changes in the transformation that has been executed to produce these models.

A *PT* can have up to three (sets of) input models. Output models produced from the original transformation are considered as input models of a *PT* because they are evolved according to the changes indicated by a *PT*. The trace model and existing input models of the original transformation are optional inputs for a *PT*. Evidently, the more information is provided as input, the more accurate the evolved output models can be. In this work, we consider the case where all three inputs are available. Thereby, a *PT* is capable of producing results, i.e., output models and trace models, that are

[1] According to [26], "*a HOT is a model transformation such that its input and/or output models are themselves transformation models*".

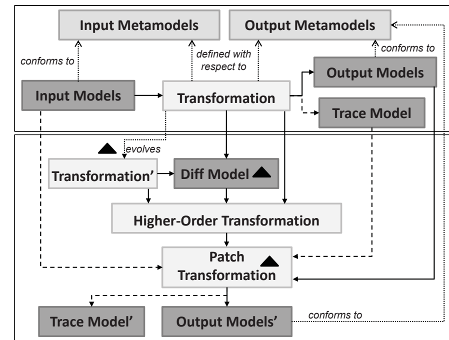

Figure 2: Patch transformation generation at a glance.

equal compared to results gained from entirely re-executing the changed transformation for which a *PT* has been generated.

3.2 Transformation Language Elements

We have selected 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, operate on read-only input models and produce write-only output models. As we will see later, this is an important property of ATL to facilitate patch transformations. In this paper, we set our focus on ATL's declarative part. The metamodel depicted in Figure 3(a) summarizes the main concepts.

A *Transformation* is composed of declarative *MatchedRules*. It gets passed *Models* as input and produces output *Models*, which conform to a metamodel (cf. Figure 3(b)). A *MatchedRule* contains one *InPattern* and one *OutPattern*. The former is a query on the input model and gathers the set of *InPatternElements* that represent the input model elements of the rule. It can also contain a *Filter*. 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*. Each *OutPatternElement* is composed of a set of *Bindings*. Their values are expressed and computed by OCL expressions that are used to initialize the features of output model elements.

Concerning the semantics of ATL, the order in which the rules are defined does not affect the computation of output models, due to a two-phase process. In the first phase, matching conditions of rules, i.e., *InPatterns*, are evaluated. Then, ATL's execution engine allocates the set of output model elements that correspond to *OutPatterns* declared in evaluated rules. In the second phase, these output model elements are initialized by feature values obtained by *Bindings*.

Listing 1 depicts an excerpt of the *Java2UML* transformation. It is currently developed and continuously revised in the ARTIST project [4]. The transformation consists of 29 rules that are defined according to the Java metamodel (*JMM*) provided by MoDisco [7] and the UML metamodel (*UMLMM*), which 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 *Bindings* 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 (input) models to the produced UML (output) models. For demonstration purposes, we assume a revision of the shown transformation that 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 on a feature of transformation languages that we call inter-rule dependencies. In our transformation, the *return type* of the value expression *s.ownedElements* is of type *Sequence(JMM!AbstractTypeElement)*. Thus, it may also contain classes as *JMM!ClassDeclaration* is a subtype of *JMM!AbstractTypeElement*. For that reason, when the *Binding* is computed, *packagedElement* will reference, among others, those elements created in the second rule[2]. It is important that we *explicitly* deal with these dependencies when creating patch transformations. In case of our transformation, a modification in the second rule may cause the re-computation of the second *Binding* in the first rule. To deal with such situations, we perform a static analysis on the revised transformation. It consists of using a HOT to determine the return types of value expressions of *Bindings*. Then, we calculate the dependencies between *Bindings* and transformation rules. As we explain in Section 3.3, we use these explicit dependencies when generating patch transformations.

Finally, we use in our approach an explicit trace metamodel (cf. Figure 3(c)). In fact, we automatically obtain a trace model from a transformation execution, e.g., by using Jouault's *TraceAdder* [14]. A *Trace* is composed of *TraceLinks*. A *TraceLink* captures the name of applied *MatchedRule* and contains *TraceElements*. These elements contain the *name* of the corresponding *InPatternElement* or *OutPatternElement* and reference to the input model elements or output model elements that have been queried or generated, respectively. To sum up, trace models explicitly capture transformation rule executions and information about input model elements that contributed to the generation of output model elements. An example trace model for a possible execution of the Java2UML transformation is shown in Figure 4.

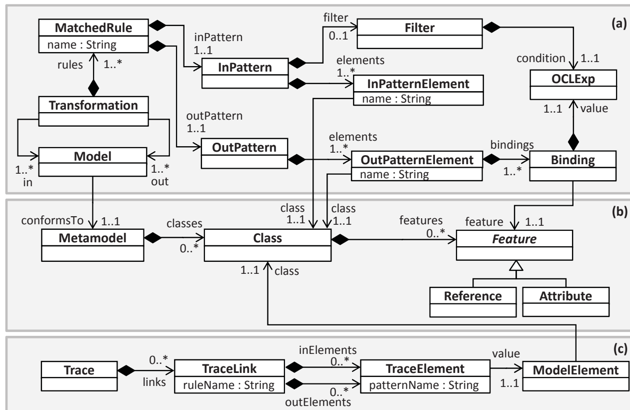

Figure 3: Metamodel excerpts: (a) transformation language, (b) metamodeling language, and (c) trace language.

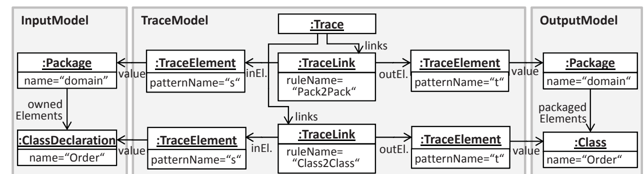

Figure 4: Example trace model fragment for Java2UML.

[2] ATL performs a transparent lookup of output model elements for given input model elements when executing *Bindings*. Hence, it automatically retrieves corresponding UML elements for queried JMM elements.

3.3 Change Types and Patch Requirements

The types of changes we have considered for the evolution of a transformation and their co-changes for existing output models are summarized in Table 1. We aim for completeness of our approach by systematically considering the addition and deletion of instances for any metaclass in the transformation metamodel (cf. Figure 3(a)) and modifications of their features.

MatchedRule. Adding or deleting a *MatchedRule* implies to add or delete the elements that the rule creates. A change of the rule name is propagated to the trace model, to keep it properly updated.

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* and we add another one, the match is realized now with the cartesian product of both element types. 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*. As a result, we delete all the changes produced by the rule and execute it. Similarly, when the *class* feature of an *InPatternElement* changes, we consider it as an addition and a deletion. As for the modification of its *name* feature, we need to propagate the change to the trace model.

Filter. The effect of adding, deleting, or modifying a *Filter* is equally treated. Even if a *Filter* is not defined, we can still consider one whose *condition* is set to *true*. Similarly, 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. In a first step, elements are added to the output model that are created from elements in the input model that now satisfy the *Filter*, whereas, in a second step, elements are deleted in the output model that correspond to elements in the input model that do not satisfy *Filter* anymore. Finally, the the corresponding *Bindings* are executed.

OutPatternElement. Adding or deleting an *OutPatternElement* implies to add or delete the respective elements in the output model and to re-execute corresponding *Bindings*. If the *class* feature is changed, we consider it as addition and deletion of the *OutPatternElement*, whereas the modification of the *name* feature is propagated to the trace model.

Binding. In case a *Binding* is added or deleted, the effect is to compute its value or delete the value that was previously computed. If the *value* expression of a *Binding* changes, it has to be recomputed and reassigned, whereas if the target *feature* is changed, we consider it as an addition and a deletion 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. If changes occur in *MatchedRules*, *InPatternElements*, *Filters* and *OutPatternElements* (except for modification of *name* features), we need to check if such changes are involved in explicit dependencies. If they are, generated patch transformations ensure that the pertinent *Bindings* are recomputed.

The way patch transformations are executed to produce evolved output models follow the same semantics ATL applies for full execution (cf. Section 3.2). In a first phase, the output model elements are appropriately added or deleted. Subsequently, in the second phase, feature values of the output model elements are computed if necessary. The latter implies the execution of affected bindings.

Listing 2 presents the patch transformation inferred from the revision of the original out-place transformation in Listing 1. It is defined in ATL's *refining mode* that enables an in-place transformation strategy [29].

Table 1: Change types and co-changes.

Concept	Change type	Co-change in output model
MatchedRule	Addition	Execution of MatchedRule
	Deletion	Deletion of previously produced elements
	Modification (*name* feature)	Propagation of name change to trace model (*ruleName* feature)
InPattern Element	Addition Deletion	1. Deletion of previously produced elements 2. Execution of the MatchedRule
	Modification (*class* feature)	Considered as Addition and Deletion of InPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName* feature)
Filter	Addition Deletion Modification (*condition* feature)	1. Deletion of elements that do not satisfy the Filter 2. Creation of elements that satisfy the Filter 3. Execution of Bindings
OutPattern Element	Addition	1. Creation of elements 2. Execution of its Bindings
	Deletion	Deletion of previously produced elements
	Modification (*class* feature)	Considered as Addition and Deletion of OutPatternElement
	Modification (*name* feature)	Propagation of name change to trace model (*patternName* feature)
Binding	Addition	Execution of added Binding
	Deletion	Deletion of feature values
	Modification (*value* feature)	Re-execution of changed Binding
	Modification (*feature* feature)	Considered as Addition and Deletion of Binding

Listing 2: Patch transformation in ATL refining mode.

```
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))))}
```

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 to 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.

3.4 Implementation

We have implemented our approach as an experimental prototype in the Eclipse environment. It is available at our project web site [20]. To compute the differences between ATL transformation versions, we first inject the transformation code into a model representation by using the ATL injector component. This model-based representation of the transformations allows us to employ EMF Compare[3] for computing the differences between them. To get a more concise diff model for our purposes, we provide some aggregation of diff elements as post-processing step of the comparison. As a result, we get a solid basis for producing patch transformations. To generate them from diff models, Xtend[4] is employed. Patch transformations are expressed in ATL code. We use the refinement mode of ATL to realize them in terms of in-place transformations. Finally, to execute the patch transformations, we use the ATL/EMF Transformation Virtual Machine [29] due to some advanced features and support for true in-place execution instead of entirely copying the input model as it is performed by the standard ATL virtual machine.

4. EVALUATION

To evaluate our approach, we investigated 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?*

RQ2: *How is the speed-up of executing patch transformations compared to re-executing the pertinent revised transformation?*

4.1 Case Study Setup

For the purpose of our case study, we selected a *Java2UML* transformation that shows an extensive and well-documented evolution. As input models for this transformation, we selected a reference application of the ARTIST project, which is based on the Java Petstore[5], and a framework that is of high relevance in this respect: EclipseLink[6]. The main rational behind the reverse-engineering of frameworks is to provide their annotations at the model level in terms of corresponding UML profiles [5]. To generate the respective Java models for the reference application and EclipseLink, we employed MoDisco. We selected six different revisions of the *Java2UML* transformation that have been performed throughout its development to cover the presented change types and the core effects of patch transformations. 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.

[3]www.eclipse.org/emf/compare

[4]www.eclipse.org/xtend

[5]www.oracle.com/technetwork/java

[6]www.eclipse.org/eclipselink

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 [20].

4.2 Case Study Results

Considering the output models of the revised transformations and the patch transformations (*RQ1*), their comparison shows that our approach produces effective results. In fact, the updates of our 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 now the focus to the runtime efficiency of our approach (*RQ2*), 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 build-up 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 an 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

	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: Re-execution vs. patch execution (time measures in sec.)

a non-invasive update to the output models is in our case study always given.

Threats to validity. We focussed on the *Java2UML* case study and applied patch transformations on small to large models that represent real-world applications and frameworks. Concerning internal validity, we need to further explore different combinations of changes and investigate if they can be correctly detected and efficiently propagated. 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 presented the problem of transformation/output model co-evolution and tackled it by reformulating changes on out-place transformations in terms of in-place patch transformations for existing output models. We demonstrated our approach for the declarative part of ATL and showed its benefits in a real-world case study. While our results are already 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 out-place transformations to match transformations. Second, we want to explore the extreme case where input models are missing for existing output models. In this respect, the question arise to which extent the output models can be kept conform to 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] C. Amelunxen, E. Legros, and A. Schürr. Generic and reflective graph transformations for the checking and enforcement of modeling guidelines. In *VL/HCC*, 2008.

[2] 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.

[3] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *SoSyM*, 11(3):431–461, 2012.

[4] 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.

[5] A. Bergmayr, M. Grossniklaus, M. Wimmer, and G. Kappel. JUMP—From Java Annotations to UML Profiles. In *MODELS*, 2014.

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

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

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

[9] 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.

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

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

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

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

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

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

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

[17] 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.

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

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

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

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

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

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

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

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

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

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

[28] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *UML*, 2004.

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

[30] 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.