Model Transformation Co-evolution:
A Semi-automatic Approach

Jokin García, Oscar Diaz, and Maider Azanza

Onekin Research Group, University of the Basque Country (UPV/EHU),
San Sebastian, Spain
{jokin.garcia,oscar.diaz,maider.azanza}@ehu.es

Abstract. Model transformations are precious and effortful outcomes of Model-Driven Engineering. As any other artifact, transformations are also subject to evolution forces. Not only are they affected by changes to transformation requirements, but also by the changes to the associated metamodels. Manual co-evolution of transformations after these metamodel changes is cumbersome and error-prone. In this setting, this paper introduces a semi-automatic process for the co-evolution of transformations after metamodel evolution. The process is divided in two main stages: at the detection stage, the changes to the metamodel are detected and classified, while the required actions for each type of change are performed at the co-evolution stage. The contributions of this paper include the automatic co-evolution of breaking and resolvable changes and the assistance to the transformation developer to aid in the co-evolution of breaking and unresolvable changes. The presented process is implemented for ATL in the CO-URE prototype.

1 Introduction

Model-Driven Engineering (MDE) describes software development approaches that are concerned with reducing the abstraction gap between the problem domain and the software implementation domain. The complexity of bridging the abstraction gap is tackled through the use of models that describe complex system at multiple levels of abstraction and from a variety of perspectives, combined with automated support for transforming and analyzing those models [6]. In this way developers can concentrate on the essence of the problem while reusing mapping strategies. Benefits include increased productivity, shorter development time, improved quality or better maintenance [15]. However, a Damocles' sword hanging over MDE is evolution. The main MDE artifacts are: *(i)* models, *(ii)* metamodels and *(iii)* transformations. While model and transformation evolution can be faced in isolation, metamodel evolution impacts models and transformations alike. Metamodel changes might have disturbing consequences on their instance models, and break apart the associated transformations. The former issue (a.k.a. model co-evolution) has been the subject of substantial work [4,9,16]. Unfortunately, transformation co-evolution has received less attention. Nevertheless, not only are transformations main enablers of the MDE advantages but their creation is programming intensive and frequently more costly

than its model counterpart [5]. This substantiates the effort to provide solid basis to assist during the transformation co-evolution effort.

Unlike previous approaches [10], we do not force to describe the evolution in terms of *ad-hoc* operands, but evolution is ascertained from differences between the original and the evolved metamodel. Next, differences are classified as [4]: *(i) Non Breaking Changes (NBC)*, i.e., changes that do not affect the transformation; *Breaking and Resolvable Changes (BRC)*, i.e., changes after which the transformations can be automatically co-evolved; and *Breaking and Unresolvable Changes (BUC)*, i.e., changes that require human intervention to co-evolve the transformation. Finally, the transformation is subject to distinct actions based on the type of the change, i.e., no action for NBC, automatic co-evolution for BRC, and assisting the user for BUC. The outcome is an evolved transformation that tackles (or warns about) the evolved metamodel. This approach is realized in the CO-URE prototype that takes as input the original Ecore metamodel, the evolved Ecore metamodel and an ATL rule transformation [11], and outputs an evolved ATL transformation. CO-URE makes intensive use of High-Order Transformations (HOTs) whereby the original transformation is handled as a model which needs to be mapped into another model (i.e. the evolved transformation). The approach can be generalized to any transformation language that provides a metamodel representation. We regard as main contributions (1) the automatic co-evolution of BRC, (2) the assistance for BUC, and (3), the CO-URE prototype.

The paper starts with a motivating scenario. Next, we outline the co-evolution process whose two main stages, detection and co-evolution, are presented in more detail in Sections 4 and 5, respectively. Section 6 introduces the CO-URE architecture and describes one of its HOT rules. Related work and conclusions end the paper.

2   Motivating Scenario

As any other software artifact, metamodels are subject to evolution. During design alternative metamodel versions may be developed. During implementation metamodels may be adapted to a concrete metamodel formalism supported by a tool. Finally, during maintenance errors in a metamodel may be corrected. Moreover, parts of the metamodel may be redesigned due to a better understanding or to facilitate reuse [22]. Simultaneously, metamodels lay at the very center of the model-based software development process. Both models and transformations are coupled to metamodels: models *conform to* metamodels, transformations *are specified upon* metamodels. Hence, metamodel evolution percolates both models and transformations. We focus on transformation co-evolution after metamodel evolution.

We use the popular *Exam2MVC* transformation [13] as a running example. This scenario envisages different types of exam questions from which Web-based exams are automatically generated along the MVC pattern [13]. Figure 1 presents the *ExamXML* metamodel and the *AssistantMVC* metamodel. The *Exam2MVC* transformation generates an *AssistantMVC* model out of an *ExamXML* model.
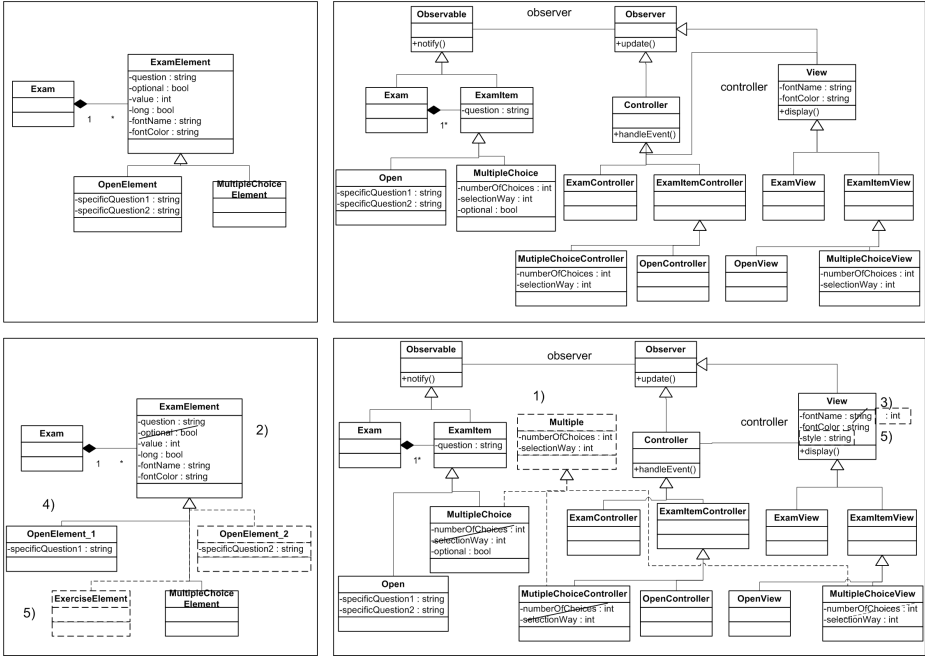
Fig. 1. *ExamXML* metamodel and *AssistantMVC* metamodel: original (above) and evolved (below)

Next, we introduce a set of evolution scenarios to be considered throughout the paper (see Figure 1):

- Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This new class abstracts away the commonality of three existing classes: *MultipleChoiceController, MultipleChoiceView* and *MultipleChoice.*
- Scenario 2. Property *optional* is deleted from *ExamXML's ExamElement.*
- Scenario 3. The *AssistantMVC's fontColor* metaproperty is changed from *string* to *integer.*
- Scenario 4. The *ExamXML's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2.*
- Scenario 5. New subclass *ExerciseElement* is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass.

The question is now how these changes impact the *Exam2MVC* transformation, better said, how can the designer be assisted in propagating these changes to the transformation counterpart. Next section outlines the process.

3   Transformation Co-evolution Process: An Outline

This section outlines the transformation co-evolution process aiming at assisting designers by automating co-evolution whenever possible. This process comprises
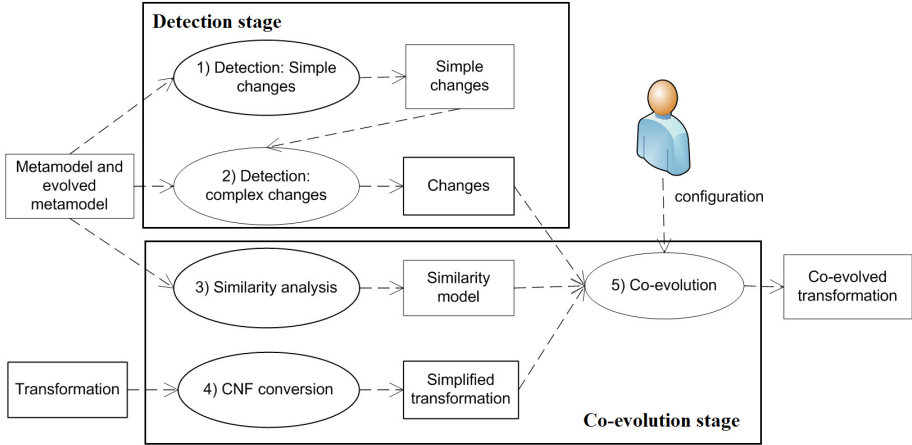
Fig. 2. Transformation co-evolution process

two main stages: *detection* and *co-evolution* (see Figure 2). Inputs include the original metamodel (M), the evolved metamodel (M') and the original transformation (T).

Detection Stage. The original metamodel and the modified metamodel are compared, and a set of differences are highlighted. Differences can range from simple cases (e.g. 'class renaming') to more complex ones (e.g. 'class splitting'). Simple changes are those that are conducted as a single shot by the user. By contrast, complex changes are abstractions over simple ones as they conform a meaningful transaction on the metamodel. Complex changes need to be treated as a unit not only from the perspective of the metamodel, but also from the co-evolution perspective. Otherwise, we risk to miss the intention of the designer when evolving the metamodel, and hence, to propagate this misunderstanding to the transformation. To this end, the detection stage includes two tasks: simple-change detection and complex-change detection. The outcome is a set of changes, both simple and complex.

Co-evolution Stage. Having a set of metamodel changes as input, this step first classifies changes based on their impact on the transformation rules. Based on the notation used in [4], we identify three types of metamodel changes:

1. *Non Breaking Changes (NBC).* These changes have no impact on the transformation. This case is illustrated by the first scenario: the introduction of the *Multiple* class as an abstraction of two existing classes. Superclass extraction has generally no impact on the transformation since metaclass properties are still reachable through inheritance. Therefore, this type of changes need to be detected, but no further action is required.
2. *Breaking and Resolvable Changes (BRC).* These changes do impact the transformation rules, but this impact is amenable to be automated. The fourth scenario is a case in point. Here, *OpenElement* is splitted into *OpenElement_1* and

```
                                        --SPLITTED RULE 1
                                        rule OpenQuestion{
                                          from  xml : ExamXML!OpenElement_1
                                          to  controller : AssistantMVC!OpenController,
                                            view : AssistantMVC!OpenView (
                                              controller <- controller,
rule OpenQuestion {                                     fontName <- 'Times',
    from xml : ExamXML!OpenElement                      fontColor <- 'Red' ),
    to  controller : AssistantMVC!OpenController(),    model : AssistantMVC!Open (
        view : AssistantMVC!OpenView(              question <- xml.question,
            controller <- controller,               specificQuestion1 <- xml.specificQuestion1,
            fontName <- 'Times',                     observers <- view )}
            fontColor <- 'Red'),            --SPLITTED RULE 2
        model : AssistantMVC!Open (          rule OpenQuestion2 {
            question <- xml.question,          from  xml : ExamXML!OpenElement_2
            specificQuestion1 <- xml.specificQuestion1,  to  controller : AssistantMVC!OpenController,
            specificQuestion2 <- xml.specificQuestion2,       view : AssistantMVC!OpenView (
            observers <- view)}                   controller <- controller,
                                                  fontName <- 'Times',
                                                  fontColor <- 'Red' ),
                                                model : AssistantMVC!Open (
                                                  question <- xml.question,
                                                  specificQuestion2 <- xml.specificQuestion2,
                                                  observers <- view )}
```
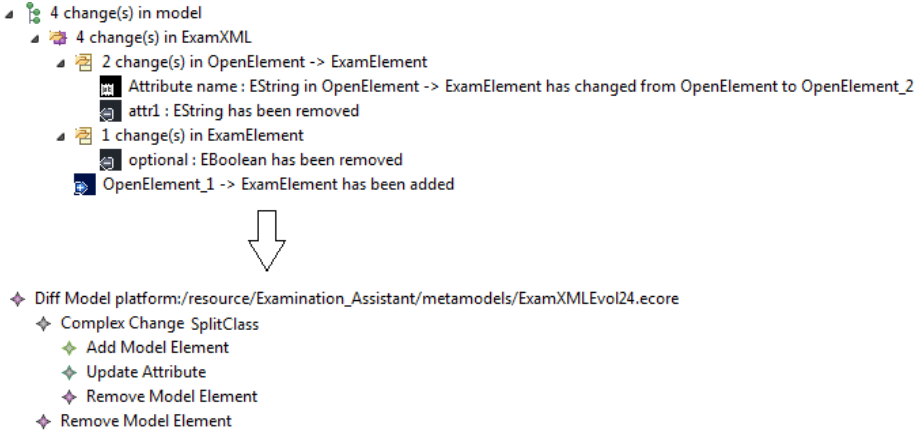
Fig. 3. *Exam2MVC* transformation: original (above), co-evolved (below)

OpenElement_2 classes. Accordingly, rules having *OpenElement* as its source might give rise to two distinct transformation rules that tackle the specifics of *OpenElement_1* and *OpenElement_2* (see Figure 3*).

3. *Breaking and Unresolvable Changes (BUC).* These changes also impact the transformation, but full automatization is not possible and user intervention is required. Reasons include: the semantics of the metamodel, the specific characteristics of the transformation language, or the specificity of the change. Hence, it will be designer's duty to manually guide the co-evolution. This is illustrated by scenario 3: *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. Type changes are the most ambiguous ones due to transformation languages being dynamically typed, and hence, susceptible to generate type errors at runtime. For instance, a rule could assign '*Times*' to *fontName*. *FontName* has now be turned into an integer, hence, making this rule inconsistent. In those cases, the option is to warn about the situation, and let the designer provide a contingency action (e.g. coming up with the *"integer"* counterpart of the formerly valid value '*Times*').

In short, for each type of change (i.e. NBC, BRC or BUC), we propose a course of action: no action, automatic transformation, and assisted transformation, respectively. To this end, the co-evolution process is complemented by two auxiliary steps: a **Conversion to Conjunctive Normal Form (CNF) step** (to address removals) and an optional **similarity analysis step** (to handle additions). Next two sections delve into the details.

4   Detection Stage

This stage takes as input both the original metamodel and the evolved metamodel, and infers the set of changes that went in between. This is achieved through two tasks: simple-change detection and complex-change detection.

▲ ⁒ 4 change(s) in model
   ▲ ⁒ 4 change(s) in ExamXML
      ▲ ⁒ 2 change(s) in OpenElement -> ExamElement
         ▦ Attribute name : EString in OpenElement -> ExamElement has changed from OpenElement to OpenElement_2
         ▩ attr1 : EString has been removed
      ▲ ⁒ 1 change(s) in ExamElement
         ▩ optional : EBoolean has been removed
      ▶ OpenElement_1 -> ExamElement has been added

⬇

◆ Diff Model platform:/resource/Examination_Assistant/metamodels/ExamXMLEvol24.ecore
   ◆ Complex Change SplitClass
      ◆ Add Model Element
      ◆ Update Attribute
      ◆ Remove Model Element
   ◆ Remove Model Element

Fig. 4. The *Difference* model (above) & *DiffExtended* model (below) for the running example. Simple changes that account for a more abstract complex change are arranged as descendants of the complex change (e.g. *AddModel*, *UpdateAttribute*, *RemoveElement* are now part of a *ComplexChange* whose *changeType* is *SplitClass*).

4.1 Simple-Change Detection

We detect simple changes as a difference between the original metamodel and the evolved metamodel. To this end, we use *EMF Compare* [19]. This tool takes two models as input and obtains the differences along the *Difference metamodel*. Back to our running example, EMF Compare is used to detect the simple changes between the original and evolved *ExamXML* metamodel as well as the original and evolved *AssistantMVC* metamodel. The output is a *Difference* model. Figure 4 (above) illustrates this *Difference* model for the *ExamXML* metamodel (scenarios 2 and 4): *UpdateAttribute*, *RemoveModelElement*, *AddModelElement* and *RemoveAttribute*. In other words, it detects that the name of the class is changed from *OpenElement* to *OpenElement_ 2*, the *specificQuestion1* metaproperty is removed from *OpenElement_ 2*, the attribute *optional* is being removed, and a new class with name *OpenElement_ 1* is added.

4.2 Complex-Change Detection

Simple changes might be semantically related to achieve a common higher-order modification. For a list of complex changes refer to [9] (we are going to analyze those relevant from the point of view the transformation co-evolution). For instance, the previous *AddModelElement* simple change hides a class split. We need to infer that a set of simple changes unitedly account for a split. Alternatively, we risk to treat each simple change on its own, which could lead to unwanted co-evolution in the transformation.

We regard complex changes as predicates over simple changes. These are auxiliary predicates needed to define them: C is the set of metaclasses and P the
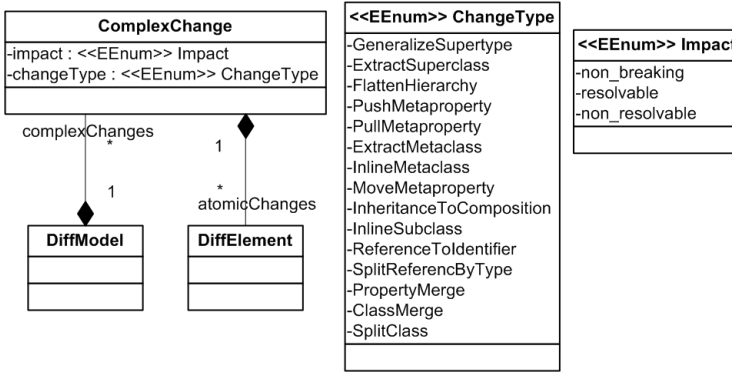
Fig. 5. *DiffExtended* metamodel: *EMFCompare*'s *Difference* metamodel is extended with the *ComplexChange* class

set of metaproperties of a metamodel. **Subclass(s: C, c: C)**: s is subclass of c; **Added_class(c: C)**: c is added to the metamodel; **Added_attribute(p: P, c: C)**: p has been added to c; **Deleted _attribute(p: P, c: C)**: p has been deleted from c; **IsAttributeOfClass(p: P, c: C)**: p belongs to c; **Added _supertype(s: C, c: C)**: supertype relationship has been added from s to c; **Deleted_supertype(s: C, c: C)**: supertype relationship has been deleted from the s to c; **Added_reference(p: P, c: C, d: D)**: Reference p from c to d is added; **Deleted_reference(p: P, c: C, d: D)**: Reference p from c to d is deleted; **Added_composition(s: C, c: C)**: composition relationship has been added from the s to the c; **Composed_name(z: string, p: string, x: string)**: delivers a new string x out of input strings z and p; **Splitted_name(c, x)** returns true if c can be obtained from x by concatenating such suffix. A notation convention exists to name split classes: the name of the original class concatenated with a number (e.g. *OpenElement_1, OpenElement_2*); **Split-ClassName(c: C)**: returns true if the new name of c is the concatenation of the old name and "_1". The list of detection predicates follows:

- **ExtractSuperclass(c:C)** iff Added_class(c) ∧ ∃p∈P, ∃s∈C
  (Added_attribute(p, c) ∧ Added_supertype(s, c) ∧ Deleted _attribute(p, s))
- **PullMetaproperty(c:C, s:C)** iff ∃p∈P (Subclass(s, c) ∧
  Added_attribute(p,c) ∧ Deleted _attribute(p, s))
- **PushMetaproperty(p: P)** iff ∃s, c∈C (Subclass(s, c) ∧
  Deleted_attribute(p,c) ∧ Added _attribute(p, s))
- **FlattenHierarchy(c:C)** iff (Deleted_class(c) ∧ ∀p∈ P |
  IsAttributeOfClass(p, c) , ∀s∈ C | Subclass(s, c) (Deleted _attribute(p,c)
  ∧ Deleted_supertype(s, c) ∧Added_attribute(p, s)))
- **MoveMetaproperty(c:C, p:P, d:C)** iff (Deleted_attribute(p,c) ∧
  Added _attribute(p, d))

- **ExtractMetaclass(c:C, d:C)** iff (Added_class(d) ∧ ∀p∈P | IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
- **InlineMetaclass(c:C, d:C)** iff (Added_class(d) ∧ Deleted_class(c) ∧ ∀p∈ P | IsAttributeOfClass(p, c) (Added _attribute(p,d) ∧ Deleted_attribute(p, c)))
- **InheritanceToComposition(c:C, d:C)** iff (Deleted_supertype(d, c) ∧ Added _composition(d, c))
- **GeneralizeSupertype(c:C, s:C, d:C)** iff (Deleted_supertype(d, s) ∧ Added _supertype(d, c) ∧Subclass(s, c))
- **InlineSubclass(c:C, d:C)** iff (Deleted_class(c) ∧ Subclass(c, d) ∧ ∀p∈P | IsAttributeOfClass(p, c) (Added _ attribute(p,d) ∧ Deleted_attribute(p, c)))
- **ReferenceToIdentifier(c:C, d:C, p:P)** iff (Deleted_reference(p, c, d) ∧ Added _attribute(p, c) ∧ Added_attribute(p, d))
- **SplitReferenceByType(c:C, d:C, x:C, s:C, p:P, y:P, z:P)** iff (Deleted_ reference(p, c, d) ∧ Added_reference(y, c, x) ∧ Added_reference(z, c, s))
- **PropertyMerge(p:P, z:P, x:P)** iff ∃c∈C (Deleted_attribute(p,c) ∧ Deleted _attribute(z, c) ∧ Added_attribute(x, c) ∧ Composed_name(z, p, x)). The last predicate delivers x by concatenating strings z and p.
- **ClassMerge(c:C, d:C)** iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Deleted_class(d) ∧ Composed _name(c, d, x))
- **SplitClass(c:C, d:C, x:C)** iff ∃y∈C (Subclass(c, y) ∧ Subclass(d, y) ∧ Added_class(d) ∧ Splitted_name(d, c)∧ SplitClassName(c)). The latter predicate needs a bit of explanation.

Implementation wise, simple changes are obtained using *EMFCompare* using the *Difference* metamodel. We propose to extend the *Difference metamodel* to account also for complex changes. Figure 5 shows an extract of the *DiffExtended* metamodel. Using the predicates aforementioned we infer complex changes that are represented as a *DiffExtended* model. Figure 4 provides a *DiffExtended* model where complex changes are also introduced. In the case that a simple change can belong to different complex changes, the biggest one has priority, e.g. *FlattenHierarchy* over *MoveMetaproperty*, as the first one includes the second.

In short, this task is realized as a transformation that takes a *Difference* model as input and obtains a *DiffExtended* model that includes both single and complex changes. Now, we are ready to percolate those changes to the transformation rules.

5   Co-evolution Stage

5.1   Similarity Analysis Step

Additional degrees of automatization can be achieved by using metamodel matching techniques. A similarity analysis is conducted between the source and

target metamodels using tools such as AML (AtlanMod Matching Language) [7]. These tools compute similarity based on the element names and the structural similarity of the metamodels. The output can be used to assist designers to fill the gaps. The approach rests on the matching effectiveness. We performed an empirical experiment based on a test-bed of 17 transformations from the ATL zoo[1]. Matching effectiveness had an average of 22-23% success (i.e., cases were an adequate binding could be suggested to the designer). This step is optional, and the weaving similarity model can be added as an input to the adaptation.

5.2   Conjunctive Normal Form Conversion Step

Rule filters are first-order predicates, normally specified using OCL. Equivalence rules of Predicate Calculus are applied to each boolean expression to get its equivalent *Conjunctive Normal Form (CNF)*, i.e., a conjunction of clauses, where a clause is a disjunction of literals (see [3] for futher details). Once in CNF, filters can be subject to "surgically removal", as explained in Subsection 5.4.

5.3   Co-evolution Step

We treat transformations as models. That is, transformations are described along a transformation metamodel. Therefore, it is possible to define (high order) transformations (HOTs) that take a transformation as input, and return a somehow modified transformation. This is precisely the approach: define correspondences that map the original transformation into an evolved transformation, taking the changes obtained during the detection stage as parameters. These HOTs are realized as ATL rules. In what follows, we summarize those rules in terms of co-evolution actions. These actions are expressed as predicates over the original transformation rules. To this end, we capture a transformation rule R as a tuple *Rule(id, source, targets, filters, mappings)* where *"source"* and *"targets"* refer to classes of the input and output metamodel, respectively; *"filters"* is a set of related predicates over the source element, such that the rule will only be triggered if the condition is satisfied; finally, *"mappings"* refer to a set of bindings to populate the attributes of the target element. A binding construct establishes the relationship between a source and a target metamodel elements. Normally, a mapping part contains a binding for each target metaclass' property. Its semantics denote what needs to be transformed into what instead of how the transformation must be performed. The left-hand side must be an attribute of the target element metaclass. The right-hand side can be a literal value, a query or an expression over the source model. Figure 3 illustrates an example of a transformation in ATL.

Transformation rules are the facts. Next, co-evolution actions are described through a set of operands and predicates over these rule facts. To avoid cluttering the description with iterations, we consider multi-valued predicates to return a single value. For instance, if a set of rules is used as parameter in the following

---

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

Bindings(r), bindings of all the rules in the set will be returned. Underscore will be used similarly to Prolog, as "don't care" variables. Predicates are intensional definitions of rule sets, and include: **RulesBySource(s)** denotes the set of rules whose source is *s;* **RulesByTarget(t)** denotes the set of rules whose target is *t;* **Binding(r, p)** returns the bindings of rule *r* which hold property *p;* **Bindings(r)** returns the bindings of rule *r;* **TargetsOfRule(r)** returns the targets of rule *r;* **FiltersOfRule(r)** returns the filters of rule *r;* **FiltersOfProperty(p)** returns the filters where the property *p* appears.

Operands act on rules: **deleteRule(r)**, which deletes the rule *r;* **deleteTarget(r, t)** which deletes target *t* from rule *r;* **deleteBinding(r, b)**, which deletes binding *b* from rule *r;* **addRule(r)**, which adds rule *r;* **addTarget(r, t)**, which adds target *t* to rule *r;* **addBinding(r, b)**, which adds a binding *b* to rule *r;* **moveTarget(r1, t, r2)**, which moves *r1*'s target *t* together with its bindings to rule *r2;* **moveBinding(r1, b, r2)** which moves *r1*'s binding *b* to *r2*, provided *r2* holds a target that matches *b*'s lefthand side; **updateFilter(r1, f1, f2)**, which updates *f1* by *f2* among *r1*'s filters; **deleteFilter(r1, f1)**, which deletes one of *r1*'s filters; **updateBinding(r1,b1, b2)**, which substitutes *r1*'s binding *b1* by *b2;* **updateSource(r, s1, s2)**, which updates source *s* of rule *r* to *s2;* **concatClass(c1, c2)**, which concats two classes names. These operands are used to specify how metamodel changes impact the transformation rules, i.e. the co-evolution actions. The list below and the list at the end of this subsection summarize the actions related to simple and complex changes, respectively.

- **removeMetaclass (c: C)**: (BRC) deleteRule(RulesBySource(c)), deleteFilter(RulesBySource(c), FiltersOfProperty(c.properties)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), c.properties))
- **removeMetaproperty(p: P)**: (BRC) deleteFilter(RulesBySource(c), FiltersOfProperty(p)), deleteBinding(RulesBySource(c), Binding(RulesBySource(c), p)). Deletions should be minimal (in Subsection 5.4)
- **updateLowerBound (p: P,NewBound)**: (NBC) No action
- **updateUpperBound (p: P,NewBound)**: (NBC) updateFilter(_, FiltersOfProperty(p), f2), updateBinding(_, Binding(_, p), b2). In case *lowerBound* converts from 1 to *, *f2* will insert a *forAll* expressions to check that all instances fulfill the condition and *b2* will use the *first()* to take the first element of the sequence. In case *lowerBound* changes from * to 1, *asSequence()* will be used in *f2* and *b2* to convert an element into a sequence.
- **updateEType**: (BUC) Syntactically right, but possible runtime type errors (refer to [18]). A warning note is generated.
- **updateESuperTypes**: (BUC) (if a metaproperty of the ancestors is accessed) Propose to copy the metaproperty of the superclass in the class.
- **updateIsAbstract (c: C,NewValue)**: (NBCor BRC) If metaclass c is turned into abstract (NewValue = "true") : Delete (Rule(c)), Delete (RHS(c)). If metaclass c is turned into a non-abstract class (NewValue = "false") then, do nothing.

- **updateELiterals (c: C)**: (BUC) Comment the structure the literal is used in, in case the user wants to use another literal. Alternative: use the default one.
- **addClass(c: C)**: (NBC) see Subsection 5.5
- **add Metaproperty(p: P)**: (NBC) see Subsection 5.5

Next, we illustrate the distinct casuistic using our running example:

- Scenario 1. The *AssistantMVC's Multiple* class is introduced in the target metamodel. This is a NBC scenario.
- Scenario 2. The property *"optional"* is deleted from *AssistantMVC's Exam-Element*. When a property is removed from the metamodel, different approaches can be taken, where the most simplistic one could be to remove the whole transformation rule where the property is used in a binding or boolean expression. However, this is a very restrictive and rather coarse-grained approach. We advocate the use of what we call the *principle of minimum deletion*, where only the part that is absolutely necessary is removed (see next subsection).
- Scenario 3. The *AssistantMVC's fontName* metaproperty is changed from *string* to *integer*. This is a BUC case.
- Scenario 4. The *AssistantMVC's OpenElement* class is splitted into *OpenElement_1* and *OpenElement_2*. As a result, rules having OpenElement as source should be co-evolved (see Figure 3). This is the case of the *OpenQuestion* rule, which is splitted in two rules: *OpenQuestion_1 and OpenQuestion_2*. The former contains the bindings related to *OpenElement_1* while the latter keeps the bindings for *OpenElement_2*.
- Scenario 5. New subclass *ExerciseElement* is added to *ExamElement* metaclass, and a new property *style* is added to *View* target metaclass. Additive evolution is a NBC case. Even though, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose we include in the co-evolution the option to generate partially new rules, as they are not fully automatable (see Subsection 5.5).

Complex Changes and their impact on transformation evolution:

- **MoveMetaproperty (c: C, p: P, d: C) if c, d $\in$ SourceClasses**: updateBinding(RulesBySource(c), Binding(RulesBySource(c), p), newBinding(p)) where newBinding works out a binding by navigating to the new location of the property, in case both classes c and d are related (navigability exists through associations). If they are not related, user assistance will be needed.
- **MoveMetaproperty (c: C, p: P, d: C) if c, d $\in$ TargetClasses**: deleteBinding(RulesByTarget(c), Binding(RulesByTarget(c), p)) or if Binding(RulesByTarget(d), p) > 0: moveBinding(RulesByTarget(c), Binding (RulesByTarget(c), p), RulesByTarget(d)).
- **FlattenHierarchy (c: C) if c $\in$ SourceClasses**: deleteRule(RulesBySource(c)) and {if RulesBySource(Subclass(c)) >0 then moveBinding(RulesBySource(c), Binding(RulesBySource(c), p), RulesBySource (subclass(c))) else addRule(rule(_, subclass(c), _, _, _))}.

- **FlattenHierarchy (c: C) if c ∈ TargetClasses**:
  deleteTarget(RulesBySource(c), c) and {if RulesByTarget(Subclass(c)>0
  then moveBinding(RulesBySource(c), Binding (RulesBySource(c), p),
  RulesBySource(subclass(c))) else addTarget (RulesBySource(subclass(c)),
  Subclass(c))}.
- **ExtractMetaclass (c: C, d: C) if c, d ∈ SourceClasses**: ad-
  dRule(rule(id, c, d, _, _)) and moveBindings(RulesBySource(c), Bind-
  ings(RulesBySource(c)), id).
- **ExtractMetaclass (c: C, d: C) if c, d ∈ TargetClasses**:
  addTarget(Rule(c, d), d) and moveBinding(RulesBySource(c), Bind-
  ings(RulesBySource(c)), addTarget(Rule(c, d), d)).
- **InlineMetaclass (c: C, d: C)**: "Extract metaclass" case and
  deleteRule(RulesBySource(c)).
- **InheritanceToComposition (c: C, d: C):** When c is the source: update-
  Filter(RulesBySource(c), FiltersOfRule(RulesBySource(c)), f2), where in f2
  refImmediateComposite() will be used in the filter. For instance: select(v |
  v.oclIsTypeOf (OpenElement))[Expression] will be converted to ExamEle-
  ment.refImmediateComposite() [Expression]. When d is the source: update-
  Binding(RulesBySource(c), Bindings(RulesBySource(c)), b2), where in *b2*
  the name of the composition relation will be introduced in the path of the
  binding. For instance, OpenElement.question [Expression] must be converted
  to OpenElement.examElement.question [Expression].
- **GeneralizeSupertype (c: C, s: C, d: C):**
  deleteBinding(RulesBySource(c), Binding (RulesBySource(c),
  Metaproperties(s))).
- **InlineSubclass (c: C, d: C):** deleteRule(RulesBySource(c)) and
  moveBinding (RulesBySource(c), Bindings(RulesBySource(c)),
  RulesBySource(d)).
- **ReferenceToIdentifier (c: C, d: C, p: P):** (As a convention, the
  id will have the same name as the deleted reference) updateBind-
  ing(RulesBySource(c), Binding (RulesBySource(c), p), newBinding), where
  the *newBinding* will replace *reference* by *metaclass.id,* e.g. if metaclass C
  with a relation p to D is converted to C with a metaproperty referring to
  the new id in D, bindings $p \leftarrow D$ (being D a reference to the generated
  element of type D) will be adapted to $p \leftarrow D.p$.
- **SplitReferenceByType (c: C, d: C, x: C, s: C, p: P, y: P, z: P):**
  deleteBinding(RulesBySource(d), p) and if x and s elements are created in
  the same rule:addBinding(RulesBySource(d), new_b).
- **PropertyMerge (p: P, z: P, x: P) if p, z, x ∈ SourceProperties:**
  updateBinding(_, Binding(_, p), newBinding), where in newBinding x is
  used instead of p or z.
- **PropertyMerge (p: P, z: P, x: P) if p, z, x ∈ TargetProperties**:
  updateBinding(_, Binding(_, p), newBinding) and deleteBinding(_, Bind-
  ing(_, z)), where newBinding will use x instead of p and z.
- **ClassMerge (c: C, d: C) if c, d ∈ Source-
  Classes:** deleteRule(RulesBySource(c)) and deleteRule (RulesBySource(d))

Table 1. Truth table for removed elements. R_T value will be interpreted as true, and R_F value as false. *L* represents a literal, which is an OCL expression that can be evaluated to a boolean value and does not include a boolean change.

L_1	L_2	L_1 AND L_2	L_1 OR L_2	NOT L_1
R_T	L_2	L_2	R_T	R_F
R_F	L_2	R_F	L_2	R_T
R_T	R_F	R_F	R_T	-
R_T	R_T	R_T	R_T	-
R_F	R_F	R_F	R_F	-

and addRule(rule(_, concatClass(c, d), union(TargetsOfRule (Rules-BySource(c)), TargetsOfRule(RulesBySource(d))), _ , union(Bindings (RulesBySource(c)), Bindings(RulesBySource(d)))). If there are filtes in the rules: updateSource(_, c, concat(c, d)) and updateSource(_, d, concat(c, d)).

– **ClassMerge (c: C, d: C) if c, d ∈ TargetClasses**: deleteTarget (RulesByTarget(c), c) and deleteTarget(RulesByTarget(d), d) and addTar-get(RulesByTarget(c), concatClass(c, d)) and addTarget(RulesByTarget(d), concatClass(c, d)).

– **SplitClass (c: C, d: C):** deleteRule(RulesBySource(c)) and addRule(rule(_, d, TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding(RulesBySource(c), Metaproperties(d)))) and addRule(rule(_, SplitClassName(c), TargetsOfRule(RulesBySource(c)), FiltersOfRule(RulesBySource(c)), Binding (RulesBySource(c), Metaproperties(SplitClassName(c)))))) .

– **PushMetaproperty (p: P):** (like move metaproperty)

5.4   The Case of the *removeProperty* Change

When a metaclass or a metaproperty is deleted, affected transformation elements have to be removed while keeping the transformation logic coherent. Coherence refers to deleting only the strictly necessary parts to prevent negative consequences. For instance, two rules might exist with complementary filters. Those filters may refer to a property. If the deletion of this property leads to the removal of the whole filter, these two rules will no longer have a discriminating filter. Therefore, the impact of metamodel element deletions should be as restrictive as possible. This is specially pressing for rule filters. This subsection discusses a way to "surgically" remove "dead" parts of rule filters. Casuistic includes:

– *Expressions with string concatenation.* This is the easiest case, let be *style ← fontName + fontColor;* an expression with the concatenation of two string metaproperties, if one of them *(e.g. fontName)* is removed, then the

expression is re-adapted to contain the rest of the metaproperties, i.e. the new expression is changed to *style ← fontColor*.

- *Expressions with creator operations of collections:* Collection types are *sets*, *ordered sets*, *bags*, and *sequences*. With expressions like *Set{London, Paris, Madrid} → union(Set{birthCity, liveCity, workCity})*, after removing a metaproperty (e.g. *birthCity*), the new expression will keep the rest of the elements, i.e *Set{London, Paris, Madrid} → union(Set{liveCity, workCity})*.
- *Expressions with other operations on collections:* There are other operations to work with collections, as *append(obj), excluding(obj), including(obj), indexOf(obj), insertAt(index, obj)*, and *prepend(obj)*. In this case, if the removed metaproperty is the parameter of the function, this part of the expression is removed. So, with an expression like *Set{London, Paris, Madrid} → append(workCity)*, after removing the *workCity* metaproperty the new expression will maintain the left hand of the expression, i.e *Set{London, Paris, Madrid}*.
- *Boolean expressions:* since a removed metaproperty cannot be evaluated, that element in the expression must be considered as undefined. Moreover, before rewriting the expression with that undefined part, it is convenient to simplify the expression as much as possible, i.e. converting it into another equivalent expression, easier to deal with. Thus, equivalence rules of *Predicate Calculus* are applied to each boolean expression to get its equivalent CNF. Inspired by [3], table 1 is proposed as truth table which defines conversion rules for CNF expressions.

  As an example, consider a metamodel with three metaproperties: *ErasmusGrant*, that says if the student has an Erasmus type grant; *speakEnglish*, that says if s/he has a good English level; and *enrolledLastYear*, that indicates if s/he is in his/her last undergraduate year. In the process of metamodel redesign, the designer could help giving some clue about the reason to take the decision of removing a metaproperty from the metamodel (*removal policy*). For example, if all students in the university had a very good level of English (because it is a new precondition for the enrollment), it could be considered as satisfied by default, and in case of removing the *speakEnglish* metaproperty, its value could be reinterpreted as *removed&satisfied-by-default* (R_T). On the other hand, if the university had decided not to participate in the Erasmus Program, no student would have such grant, and in case of removing the *ErasmusGrant* metaproperty, its value could be reinterpreted as *removed&unsatisfied-by-default* (R_F).

  If, in the previous example, there had been this expression *not ErasmusGrant or (speakEnglish and enrolledLastYear)*, and later the redesign process decided to remove the *speakEnglish* metaproperty, then according to the truth table the expression would be rewritten as *not ErasmusGrant or enrolledLastYear*; if the removed metaproperty had been *ErasmusGrant* with R_F policy, then the new expression would have been *true*.
- *Expressions with loop operations:* In ATL the syntax used to call an iterative expression is the following: *source → operation_name (iterators | body)*. Among these operations there are *any(expr), collect(expr), exists(expr),*

```
rule MultipleChoice {
  from  xml : ExamXML!MCElement
  to  controller : AssistantMVC!MultipleChoiceController,
    view : AssistantMVC!MultipleChoiceView (
      controller <- controller,
      fontName <- xml.attr1,
      fontColor <- xml.attr2  ),
      --fill right part
      style <- xml.style  )}

--NEW RULE
rule ExerciseElement {
  from   s : ExamXML!ExerciseElement
  to   t : AssistantMVC!Exam (
      --write the bindings
      "--target" <- s.source
      )}
```

Fig. 6. Generated skeletons for the new *style* property (above) and the new *ExerciseElement* class (below)

forAll(expr), one(expr), select(expr), and so on. For instance, in *self.items* → *exists(i | i.question.size()>50)*, if the removed metaproperty (e.g. *items*) takes part in the source, the whole expression is removed, but if the removed metaproperty takes part in the body, the rules for the boolean expressions must be applied.

– For more ambiguous cases, we resort to reporting the ambiguity and letting the designer decide. For instance, if the returned type of a helper is removed, the helper cannot be considered during binding, and a warning note is introduced. Or if the removal of a property makes the scope of two rules coincide then, the first one is commented.

Back to our second scenario (i.e. removal of *optional* from *ExamElement*), consider we have two rules whose filters refer to *optional*:

– *(value>5 and optional) or long.* Applying equivalences from table 1, the evolved filter becomes *(value > 5 or long)*
– *not ((value>5 and optional) or long).* Using Morgan's laws, its CNF counterpart is: *(not value>5 or not optional) and not long.* Applying equivalences from table 1, the evolved filter results in *(not value>5 and not long).*

In this way, "surgical removal" permits to limit the impact of deletion of properties in the associated rules.

5.5   The Case of *addClass* and *addProperty* Changes

Although additive evolution is considered *NBC*, it is not unusual to need new rules or bindings to maintain the metamodel coverage level. For this purpose, we include in the co-evolution the option to generate partially new rule skeletons. Our fifth scenario illustrates this situation: addition of the *ExerciseElement* metaclass, and addition of the *style* property to the *View* metaclasss. The engineered co-evolution can be seen at work in Figure 6: a rule is partially generated
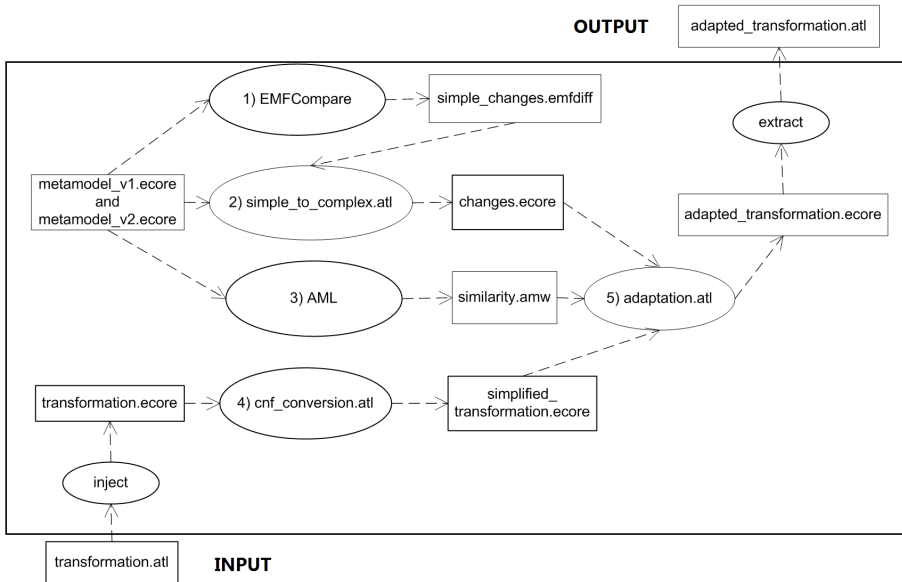
Fig. 7. CO-EUR architecture

to tackle the addition of a new source metaclass while a new partial binding is proposed to address new properties. In the latter case, only a simple binding is generated (e.g. *target_ metaproperty ← source_ metaproperty*) which needs to be completed by the designer (in the example, *xml.style*).

6   Implementation

The CO-EUR prototype is available[2] as a proof-of-concept of the feasibility of this approach for ATL rule co-evolution. Figure 7 depicts the main CO-EUR modules that mimic the co-evolution workflow introduced in Section 2. CO-EUR takes an ATL file (.atl), two Ecore metamodels (.ecore) as input, and returns an ATL file that tackles the differences between the input Ecore models.

The main effort was devoted to the adaptation module. Implementation wise, this module also represents the main innovative approach since transformation co-evolution is achieved using HOT transformations. Along the MDE *motto*: "*everything is a model*" [2], transformations are models that conform to their own metamodel (i.e., the transformation language). Being models, *(Higher Order)* transformations can be used to map the original transformation model into a co-evolved transformation model that caters for the metamodel changes. Figure 8 outlines one such HOT transformation that tackles the *splitClass* case. The pattern includes: a "main" rule, some lazy rules that are called from it

---

[2] `www.onekin.org/downloads/public/examination-assistant.rar`

```
1 helper def : deleteRule_Splitclass (param : Sequence(ATL!MatchedRule)) : Sequence(ATL!MatchedRule) =
2     let elements : Sequence(String) = self.getSplittedClasses
3     in elements->iterate(p; y : Sequence(ATL!MatchedRule) = param |
4         if self.contains(p, param) then
5             self.deleteRule_Splitclass(y->excluding(param->at(self.index(p, param))))
6         else
7             y
8         endif);
9 rule Module_Splitclass {
10    from    s : ATL!Module(
11        self.getUpdateAttributeRight_Splitclass.size()>0
12    to  t : ATL!Module (
13            elements <- self.deleteRule_Splitclass(s.elements)
14        )
15    do{
16    t.elements <- t.elements->append(thisModule.MatchedRule2MatchedRule_Splitclass(t.elements));
17    t.elements <- t.elements->append(thisModule.MatchedRule2MatchedRule2_Splitclass(t.elements));}}
18 lazy rule MatchedRule2MatchedRule_Splitclass {
19    from    s : ATL!MatchedRule
20    to  mr : ATL!MatchedRule ()
21        [...]
22    do{
23    for (iterator in self.simpleOutPatternElements){
24    op_i_c2.elements <- op_i_c2.elements->append(thisModule.SOPE2SOPE_Splitclass(op_i_c2.elements));
25    self.index_Splitclass <- self.index_Splitclass + 1;}}}
26 lazy rule SOPE2SOPE_Splitclass {
27    from    s : ATL!SimpleOutPatternElement
28    to  ope_i_c2 : ATL!SimpleOutPatternElement()
29    do{
30    self.indexBinding_Splitclass <- 1;
31    for (iterator in self.simpleOutPatternElements.at(self.index).bindings){
32        if (self.simpleOutPatternElements.at(self.index_Splitclass).bindings.at
33            (self.indexBinding_Splitclass).value.oclIsTypeOf(ATL!VariableExp)){
34            ope_i_c2.bindings <- ope_i_c2.bindings->append(self.B2B_Splitclass(ope_i_c2.bindings));
35        }else{
36            if (self.simpleOutPatternElements.at(self.index_Splitclass).bindings.at
37                (self.indexBinding_Splitclass).value.oclIsTypeOf(ATL!StringExp)){
38                ope_i_c2.bindings <- ope_i_c2.bindings->
39                append(self.B2BString_Splitclass(ope_i_c2.bindings));}
40        self.indexBinding_Splitclass <- self.indexBinding_Splitclass + 1;}}}
41 lazy rule B2B_Splitclass {
42    from    s : ATL!Binding
43    to  b : ATL!Binding (
44        [...]}}
```

Fig. 8. HOT rules to cope with class split. HOTs' input and output models conform to the ATL metamodel.

to create elements, and some helpers to modularize the functionality. In this specific case, *Module_Splitclass* is the "main" rule (line 9), which will be executed when there is any change of type *Splitclass*. In the *to part* of the rule, the helper *deleteRule_Splitclass* is called (line 13) which causes the deletion of the rule referring to the deleted metaclass. Then, the imperative part of the rule (*do*) creates two new rules: *MatchedRule2MatchedRule_Splitclass* and *MatchedRule2MatchedRule2_Splitclass* (lines 16 and 17). These rules in turn refer to rule *SOPE2SOPE_Splitclass* (line 24), which creates a *SimpleOutPatternElement* for the new generated rules. Finally, this rule invokes *B2B_Splitclass* to geneate the bindings (lines 34 and 39).

7   Related Work

Although co-evolution of models after metamodel evolution has been widely studied [4,9,16], transformations have raised less attention. A lot of research has

been carried out in the model co-evolution area and some proposals have been done to semi-automatically adapt models to metamodel evolution. Three main strategies have been used [10]: *(i)* manual specification: these approaches provide transformation languages to manually specify the migration (e.g. [16]); *(ii)* matching approaches: they intend to automatically derive a migration from the matching between two metamodel versions (e.g. [4]); and *(iii)* co-evolution based on operators: they record the coupled operations which are used to evolve the metamodel and which also encapsulate a model migration (e.g. [9]). Following this classification, our approach would be in the second type, as we do not know changes in advance or make them in any specific tool. But on the other hand, we rely our complex changes in a taxonomy of operators based on the third type ([9]). Our approach is similar, as each change has an associated co-evolution, but the difference is that we do not create explicitly the operators, as they are automatically derived. In some cases changes in metamodels do not affect transformations, as studied in [18], where authors conclude that the addition of new classes and broadening of multiplicity constraints do not break the subtyping relationship between metamodel versions. But often changes do have an impact on transformations. To the best of our knowledge, two authors ([14] and [17]) have dealt with transformation co-evolution. The first case is limited to graph-based languages, considering simple changes and considering subtractive changes only as coarse-grained removals (i.e., rule level deletions). In contrast, we focus on rule-based declarative languages, deleting as little as possible, and considering complex changes. In [17] authors explain a fundamental idea, e.g., the convenience of using operators in the co-evolution of transformations. Compared to this publication, our contribution would be an automatic conversion from simple to complex changes, minimum deletion and an implementation of co-evolutions in ATL. First issue of the approach, the conversion of simple to complex changes is treated in [8] and [21]. The former is based on a DSL for expressing model matching and the later uses a sequence of operator instances as evolution trace, and they allow to make changes over changes.

Our co-evolution process only guarantees that the transformation is syntactically correct, and if other correctness properties need to be checked, other complementing works will have to be considered, as analysis and simulation [1], testing [12] or metamodel coverage [23]. In the case where co-evolution is done manually, coverage analysis can be used to determine whether the changes to a metamodel affect the transformation [20].

8   Conclusions

We addressed how metamodel evolution can be semi-automatically propagated to the transformation counterpart. The process flow includes: (1) detecting simple changes from differences between the original metamodel and the evolved metamodel, (2) deriving complex changes from simple changes, (3) translating boolean expressions to the CNF form, (4) if available, capitalize on model similarity, and finally, (5) map the original transformation into an evolved transformation that (partially) tackles the evolved metamodel. The approach is realized

for EMOF/ Ecore-based metamodels, and ATL transformations. The approach relieves domain experts from handling routine cases so that they can now focus on the more demanding scenarios (e.g. additive evolution). The use of high-level transformations implies the existence of a transformation metamodel. So far this is available for main transformation languages such as ATL or RubyTL.

References

1. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Baudry, B., Faivre, A., Ghosh, S., Pretschner, A. (eds.) Proceedings of the 4th MoDeVVa Workshop Model-Driven Engineering, Verification and Validation, pp. 47–56 (2007), http://kyriakos.anastasakis.net/prof/pubs/modevva07.pdf
2. Bézivin, J.: In Search of a Basic Principle for Model-Driven Engineering. UP-GRADE, The European Journal for the Informatics Professional, Special Issue on UML and Model Engineering 5(2), 21–24 (2004)
3. Cabot, J., Conesa, J.: Automatic Integrity Constraint Evolution due to Model Sub-tract Operations. In: Wang, S., Tanaka, K., Zhou, S., Ling, T.-W., Guan, J., Yang, D.-Q., Grandi, F., Mangina, E.E., Song, I.-Y., Mayr, H.C. (eds.) ER Workshops 2004. LNCS, vol. 3289, pp. 350–362. Springer, Heidelberg (2004)
4. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Enterprise Distributed Object Computing Conference (2008)
5. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is Needed for Managing Co-evolution in MDE? In: Proc. of the 2nd International Workshop on Model Comparison in Practice, IWMCP 2011, pp. 30–38. ACM, New York (2011)
6. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: Workshop on the Future of Software Engineering (FOSE 2007), at the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA, pp. 37–54 (2007)
7. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proc. of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM 2009) (2009)
8. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing Model Adaptation by Precise Detection of Metamodel Changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009)
9. Herrmannsdoefer, M., Vermolen, S., Wachsmuth, G.: An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In: Software Language Engineering, Third International Conference, Software Language Engineering 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers (2011)
10. Herrmannsdoerfer, M.: COPE – A Workbench for the Coupled Evolution of Meta-models and Models. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 286–295. Springer, Heidelberg (2011)

11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming (SCP) 72(1-2), 31–39 (2008)
12. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations – First Experiences Using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006 Workshops. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
13. Kurtev, I.: Adaptability of Model Transformations, ch. 5. PhD thesis, University of Twente, Enschede (May 2005)
14. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 23–41. Springer, Heidelberg (2010)
15. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
16. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
17. Schätz, B., Deridder, D., Pierantonio, A., Sprinkle, J., Tamzalit, D.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In: Proc. of the International Workshop on Models and Evolution (ME 2011) at MoDELS 2011, pp. 54–63 (2010)
18. Steel, J., Jézéquel, J.: On Model Typing. Software and System Modeling 6(4), 401–413 (2007)
19. Toulmé, A.: Presentation of EMF Compare Utility. In: Eclipse Modeling Symposium 2006, pp. 1–8 (2006)
20. van Amstel, M.F., van den Brand, M.G.J.: Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 108–122. Springer, Heidelberg (2011)
21. Vermolen, S.D., Wachsmuth, G., Visser, E.: Reconstructing Complex Metamodel Evolution. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 201–221. Springer, Heidelberg (2012)
22. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
23. Wang, J., Kim, S., Carrington, D.: Verifying Metamodel Coverage of Model Transformations. In: Proc. of the Australian Software Engineering Conference, pp. 270–282. IEEE Computer Society, Washington, DC (2006)