Automating Co-evolution in Model-Driven Engineering*

Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, Alfonso Pierantonio
Dipartimento di Informatica
Università degli Studi di L'Aquila
I–67100 L'Aquila, Italy
Email: {cicchetti, diruscio, eramo, alfonso}@di.univaq.it

Abstract

Software development is witnessing the increasing need of version management techniques for supporting the evolution of model-based artefacts. In this respect, metamodels can be considered one of the basic concepts of Model-Driven Engineering and are expected to evolve during their life-cycle. As a consequence, models conforming to changed metamodels have to be updated for preserving their well-formedness.

This paper deals with the co-adaptation problems by proposing higher-order model transformations which take a difference model recording the metamodel evolution and produce a model transformation able to co-evolve the involved models.

1 Introduction

Model-Driven Engineering (MDE) [19] aims at rendering business logic and intellectual property resilient to technological changes by shifting the focus of software development from coding to modeling. In general, domains are analysed and engineered by means of a *metamodel*, i.e. a coherent set of interrelated concepts. A model is said to *conform* to a metamodel, or in other words it is expressed by the concepts encoded in the metamodel, constraints are expressed at the metalevel, and model transformation occurs when a source model is modified to produce a target model.

Evolution is an inevitable aspect which affects the whole life-cycle of software systems [12]. In general, artefacts can be subject to many kinds of changes, which range from requirements through architecture and design, to source code, documentation and test suites. Moreover, taxonomies of software evolution distinguish maintenance activities on the basis of their purpose (i.e. updative, adaptive, performance, corrective or reductive) or technical aspects (i.e., the when, where, what and how of software changes) [14, 5]. Therefore, evolution management is a complex task which requires specialized discipline and tool support.

Similarly to other software artefacts, metamodels can evolve over time too [9]. Accordingly, models need to be *co-adapted*[1] in order to remain compliant to the metamodel and not become eventually invalid. When manually operated the adaptation is error-prone and can give place to inconsistencies between the metamodel and the related artefacts. Such issue becomes very relevant when dealing with enterprise applications, since in general system models encompass a large population of instances which need to be appropriately adapted, hence inconsistencies can possibly lead to irremediable information erosion [23].

This work proposes a transformational approach to model co-evolution, i.e. how to automatically generate well-defined adaptation steps directly from the modifications the metamodel underwent. In particular, the approach is based on a model difference representation [7] which is used to specify in a *difference* model the metamodel changes. Thus, the co-adaptation is given as a higher-order model transformation which takes the difference model recording the metamodel evolution and generates a model transformation able to produce the co-evolution of models. Especially, the proposal shows how the *breaking resolvable* and *unresolvable* changes (see Sect. 2) require a specific management whenever interdependencies among them occur.

The structure of the paper is as follows. In Sect. 2 the different kinds of modifications a metamodel can be subject to are illustrated and categorized in accordance with the available literature. Moreover, it presents the typologies of co-adaptation steps a metamodel evolution induces. Then, the proposed approach is described: Sect. 3 introduces a model-based representation of the metamodel evolution, whereas Sect. 4 describes the automated co-adaptation. Finally, in Sect. 5 and Sect. 6 related works and some conclusions are

*Partially supported by the European Communitys 7th Framework Programme (FP7/2007-2013), grant agreement n. 214898.

[1] The terms (co-)adaptation and (co-)evolution will be used as synonyms throughout the paper.

discussed, respectively.

2 Metamodel evolution and model co-evolution

Metamodels can be considered one of the constituting concepts of MDE, since they are the formal definition of well-formed models, or in other words they constitute the languages by which a given reality can be described in some abstract sense [2]. Metamodels are expected to evolve during their life-cycle, thus causing possible problems to existing models which conform to the old version of the metamodel and do not conform to the new version anymore. The problem is due to the incompatibility between the metamodel revisions and a possible solution is the adoption of mechanisms of model co-evolution, i.e. models need to be migrated in new instances according to the changes of the corresponding metamodel.

Unfortunately, model co-evolution is not always simple and presents intrinsic difficulties which are related to the kind of evolution the metamodel has been subject to. Going into more details, metamodels may evolve in different ways: some changes may be additive and independent from the other elements, thus requiring no or little instance revision. In other cases metamodel manipulations introduce incompatibilities and inconsistencies which can not be easily (and automatically) resolved.

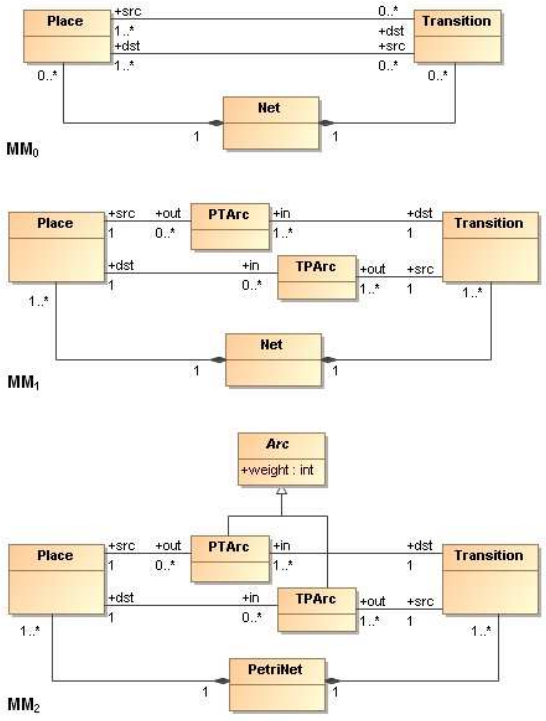

Figure 1. Petri Net metamodel evolution

In Fig. 1 it is depicted an example of the evolution of a (simplified) Petri Net metamodel, which takes inspiration from the work in [23]. The initial Petri Net (MM_0) consists of `Place`s and `Transition`s; moreover, places can have source and/or destination transitions, whereas transitions must link source and destination places (`src` and `dst` association roles, respectively). In the new metamodel MM_1, each `Net` has at least one `Place` and one `Transition`. Besides, arcs between places and transitions are made explicit by extracting `PTArc` and `TPArc` metaclasses. This refinement permits to add further properties to relationships between places and transitions. For example, the Petri Net formalism can be extended by annotating arcs with weights. As `PTArc` and `TPArc` both represent arcs, they can be generalized by a superclass, and a new integer metaproperty can be added in it. Therefore, an abstract class `Arc` encompassing the integer metaproperty `weight` has been added in MM_2 revision of the metamodel. Finally, `Net` has been renamed into `PetriNet`. The metamodels in Fig. 1 will be exploited as the running example throughout the paper. They have been kept deliberately simple because of space limitations, even though they are suitable to present all the insights of the co-adaptation mechanisms as already demonstrated in [23].

The revisions illustrated so far can invalidate existing instances; therefore, each version needs to be analysed to comprehend the various kind of updates it has been subject to and, eventually, to elicit the necessary adaptations of corresponding models. Metamodel manipulations can be classified by their corrupting or non-corrupting effects on existing instances [11]:

- *non-breaking changes*: changes which do not break the conformance of models to the corresponding metamodel;
- *breaking and resolvable changes*: changes which break the conformance of models even though they can be automatically co-adapted;
- *breaking and unresolvable changes*: changes which break the conformance of models which can not automatically co-evolved and user intervention is required.

In other words, *non-breaking changes* consist of additions of new elements in a metamodel `MM` leading to MM' without compromising models which conform to `MM` and thus, in turn, conform to MM'. For instance, in the metamodel MM_2 illustrated in Fig. 1 the abstract metaclass `Arc` has been added as a generalization of the `PTArc` and `TPArc` metaclasses (without considering the new attribute `weight`). After such a modification, models conforming to MM_1 still conform to MM_2 and co-evolution is not necessary. Unfortunately, this is not always the case since in general changes may break models even though sometimes automatic resolution can be performed, i.e. when facing *breaking and*

resolvable changes. For instance, the Petri Net metamodel MM_1 in Fig. 1 is enriched with the new `PTArc` and `TPArc` metaclasses. Such a modification breaks the models that conform to MM_0 since according to the new metamodel MM_1, `Place` and `Transition` instances can not be directly related, but `PTArc` and `TPArc` elements are required. However, models can be automatically migrated by adding for each couple of `Place` and `Transition` entities two additional `PTArc` and `TPArc` instances between them.

Often manual interventions are needed to solve breaking changes like, for instance, the addition of the new attribute `weight` to the class `Arc` of MM_2 in Fig. 1 which were not specified in MM_1. The models conforming to MM_1 can not be automatically co-evolved since only a human intervention can introduce the missing information related to the weight of the arc being specified, or otherwise default values have to be considered. We refer to such situations as *breaking and unresolvable changes*.

All the scenarios of model co-adaptations can be managed with respect to the possible metamodel modifications which can be distinguished into *additive*, *subtractive*, and *updative*. In particular, with additive changes we refer to metamodel element additions which in turn can be further distinguished as follows:

– *Add metaclass*: introducing new metaclasses is a common practice in metamodel evolution which gives place to metamodel extensions. Adding new metaclasses raises co-evolution issues only if the new elements are mandatory with respect to the specified cardinality. In this case, new instances of the added metaclass have to be accordingly introduced in the existing models;

– *Add metaproperty*: this is similar to the previous case since a new metaproperty may be or not obligatory with respect to the specified cardinality. The existing models maintain the conformance to the considered metamodel if the addition occurs in abstract metaclasses without subclasses; in other cases, human intervention is required to specify the value of the added property in all the involved model elements;

– *Generalize metaproperty*: a metaproperty is generalized when its multiplicity or type are relaxed. For instance, if the cardinality `3..n` of a sample metaclass `MC` is modified in `0..n`, no co-evolution actions are required on the corresponding models since the existing instances of `MC` still conform to the new version of the metaclass;

– *Pull metaproperty*: a metaproperty `p` is pulled in a superclass `A` and the old one is removed from a subclass `B`. As a consequence, the instances of the metaclass `A` have to be modified by inheriting the value of `p` from the instances of the metaclass `B`;

– *Extract superclass*: a superclass is extracted in a hierarchy and a set of properties is pulled on. If the superclass is abstract model instances are preserved, otherwise the effects are referable to metaproperty pulls.

Change type	Change
Non-breaking changes	Generalize metaproperty
	Add (non-obligatory) metaclass
	Add (non-obligatory) metaproperty
Breaking and resolvable changes	Extract (abstract) superclass
	Eliminate metaclass
	Eliminate metaproperty
	Push metaproperty
	Flatten hierarchy
	Rename metaelement
	Move metaproperty
	Extract/inline metaclass
Breaking and unresolvable changes	Add obligatory metaclass
	Add obligatory metaproperty
	Pull metaproperty
	Restrict metaproperty
	Extract (non-abstract) superclass

Table 1. Changes classification

Subtractive changes consist of the deletion of some of the existing metamodel elements as described in the following:

– *Eliminate metaclass*: a metaclass is deleted by giving place to a sub metamodel of the initial one. In general, such a change induces in the corresponding models the deletions of all the metaclass instances. Moreover, if the involved metaclass has subclasses or it is referred by other metaclasses, the elimitation causes side effects also to the related entities;

– *Eliminate metaproperty*: a property is eliminated from a metaclass, it has the same effect of the previous modification;

– *Push metaproperty*: pushing a property in subclasses means that it is deleted from an initial superclass `A` and then cloned in all the subclasses `C` of `A`. If `A` is abstract then such a metamodel modification does not require any model co-adaptation, otherwise all the instances of `A` and its subclasses need to be accordingly modified;

– *Flatten hierarchy*: to flatten a hierarchy means eliminating a superclass and introducing all its properties into the subclasses. This scenario can be referred to metaproperty pushes;

– *Restrict metaproperty*: a metaproperty is restricted when its multiplicity or type are enforced. It is a complex case where instances need to be co-adapted or restricted. Restricting the upper bound of the multiplicity requires a selection of certain values to be deleted. Increasing the lower bound requires new values to be added for the involved element which usually are manually provided. Restricting the type of a property requires type conversion for each value.

Finally, a new version of the model can consist of some updates of already existing elements leading to updative modifications which can be grouped as follows:

– *Rename metaelement*: renaming is a simple case in which the change needs to be propagated to existing instances and can be performed in an automatic way;
– *Move metaproperty*: it consists of moving a property `p` from a metaclass `A` to a metaclass `B`. This is a resolvable change and the existing models can be easily co-evolved by moving the property `p` from all the instances of the metaclass `A` to the instances of `B`;
– *Extract/inline metaclass*: extracting a metaclass means to create a new class and move the relevant fields from the old class into the new one. Vice versa, to inline a metaclass means to move all its features into another class and delete the former. Both metamodel refactorings induce automated model co-evolutions.

The classification illustrated so far is summarized in Tab. 1 and makes evident the fundamental role of evolution representation. At a first glance it seems that the classification does not encompass *references* that are associations amongst metaclasses. However, references can be considered properties of metaclasses at the same level of attributes.

Metamodel evolutions can be precisely categorized by understanding the kind of modifications a metamodel undergone. Moreover, starting from the classification it is possible to adopt adequate countermeasures to co-evolve existing instances. Nonetheless, it is worth noting that the classification summarized in Tab. 1 is based on a clear distinction between the metamodel evolution categories. Unfortunately, in real world experiences the evolution of a metamodel can not be reduced to a sequence of atomic changes, generally several types of changes are operated as affecting multiple elements with different impacts on the co-adaptation. Furthermore, the entities involved in the evolution can be related one another. Therefore, since co-adaptation mechanisms are based on the described change classification, a metamodel adaptation will need to be decomposed in terms of the induced co-evolution categories. The possibility to have a set of dependences among the several parts of the evolution makes the updates not always distinguishable as single atomic steps of the metamodel revision, but requires a further refinement of the classification as introduced in the next section and discussed in details in Sect. 4.

3 Formalizing metamodel differences

The problem of model differences is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [13]. Recently, in [7, 18] two similar techniques have been introduced to represent differences as models, hereafter called *difference models*; interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches. In the rest of the section, we recall the difference representation approach defined in [7] in order to provide the reader with the technical details which underpin the solution proposed in Sect. 4.

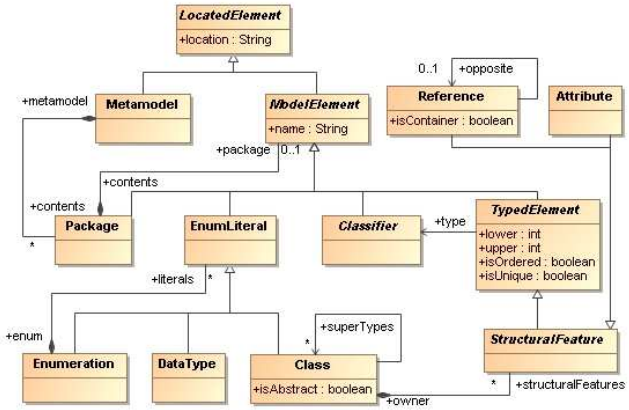

Figure 2. KM3 metamodel

Despite the work in [7] has been introduced to deal with model revisions, it is easily adaptable to metamodel evolutions too. In fact, a metamodel is a model itself, which conforms to a metamodel referred to as the meta metamodel [2]. For presentation purposes, the KM3 language in Fig. 2 is considered throughout the paper, even though the solution can be generalized to any metamodeling language like OMG/MOF [15] or EMF/Ecore [4].

The overall structure of the change representation mechanism is depicted in Fig. 3: given two *base metamodels* $\texttt{MM}_1$ and $\texttt{MM}_2$ which conform to an arbitrary *base meta metamodel* (KM3 in our case), their difference conforms to a *difference metamodel* `MMD` derived from KM3 by means of an automated transformation `MM2MMD`. The base meta metamodel, extended as prescribed by such a transformation, consists of new constructs able to represent the possible modifications that can occur on metamodels and which can be grouped as follows:

– *additions*: new elements are added in the initial metamodel; with respect to the classification given in Sect. 2, *Add metaclass* and *Extract superclass* involve this kind of change;
– *deletions*: some of the existing elements are deleted as a whole. *Eliminate metaclass* and *Flatten hierarchy* fall in this category of manipulations;
– *changes*: a new version of the metamodel being considered can consist of updates of already existing elements. For instance, *Rename metaelement* and *Restrict metaproperty* require this type of modification. Also the addition and deletion of metaproperty (i.e. *Add*

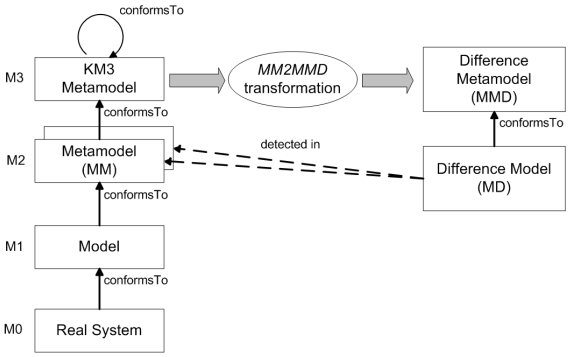

Figure 3. Overall structure of the model difference representation approach

metaproperty and *Eliminate metaproperty*, respectively) are modelled through this construct. In fact, when a metaelement is included in a container the manipulation is represented as a *change* of the container itself.

In order to represent the differences between the Petri Net metamodel revisions, the extended KM3 meta metamodel depicted in Fig. 4 is generated by applying the `MM2MMD` transformation in Fig. 3 previously mentioned. For each metaclass `MC` of the KM3 metamodel, the additional metaclasses `AddedMC`, `DeletedMC`, and `ChangedMC` are generated. For instance, the metaclass `Class` in Fig. 2 induces the generation of the metaclasses `AddedClass`, `DeletedClass`, and `ChangedClass` as depicted in Fig. 4. In the same way, `Reference` metaclass induces `AddedReference`, `DeletedReference`, and `ChangedReference`.

The generated difference metamodel is able to represent all the differences amongst metamodels which conform to KM3. For instance, the model in Fig. 5 conforms to the generated metamodel in Fig. 4 and represents the differences between the Petri Net metamodels specified in Fig. 1. The differences depicted in such a model can be summarized as follows:

1) the addition of the new class `PTArc` in the $\texttt{MM}_1$ revision of the Petri Net metamodel is represented by means of an `AddedClass` instance, as illustrated by model difference $\Delta_{0,1}$ in Fig. 5. Moreover, the reference between `Place` and `Transition` named `dst` has been updated to link `PTArc` with name `out`. Analogously, the reverse reference named `src` has been manipulated to point `PTArc` and named as `in`. Two new references have been added through the corresponding `AddedReference` instances to realize the reverse links from `PTArc` to `Place` and `Transition`, respectively. Finally, the composition relationship between `Net` and `Place` has been updated by prescribing the existence of at least one `Place` through the `lower` property which has been updated from `0` to `1`. The same enforcement has been done to the composition between `Net` and `Transition`;

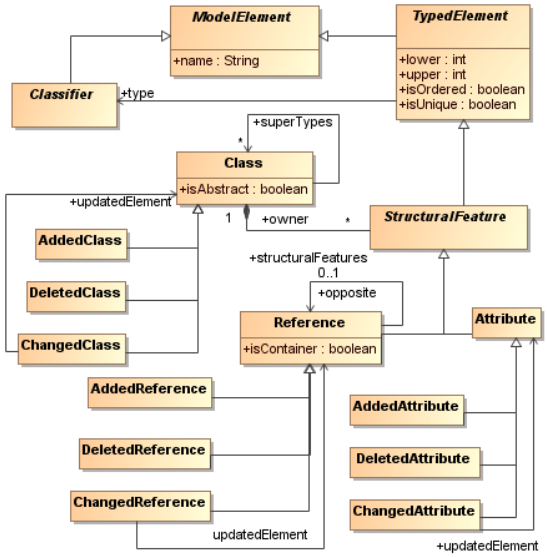

Figure 4. Generated difference KM3 metamodel

2) the addition of the new abstract class `Arc` in $\texttt{MM}_2$ together with its attribute `weight` is represented through an instance of the `AddedClass` and the `AddedAttribute` metaclasses in the $\Delta_{1,2}$ delta of Fig. 5. In the meanwhile, `PTArc` and `TPArc` classes are made specializations of `Arc`. Finally, `Net` entity is renamed as `PetriNet`.

Difference models like the one in Fig. 5 can be obtained by using today's available tools like EMFCompare [21] and SiDiff [22], which are not discussed here due to space limitation.

The representation mechanism used so far allows to identify changes which occurred in a metamodel revision and satisfies a number of properties, as illustrated in [7]. One of them is the *compositionality*, i.e. the possibility to combine difference models in interesting constructions like the sequential and the parallel compositions, which in turn result in valid difference models themselves. For the sake of simplicity, let us consider only two modifications over the initial model: the sequential composition of such manipulations corresponds to merging the modifications conveyed by the first document and then, in turn, by the second one in a resulting difference model containing a minimal difference set, i.e., only those modifications which have not been overridden by subsequent manipulations. Whereas, parallel compositions are exploited to combine modifications operated from the same ancestor in a concurrent way. In case both manipulations are not affecting the same elements they are said *parallel independent* and their composition is obtained by merging the difference models by interleaving the single changes and assimilating

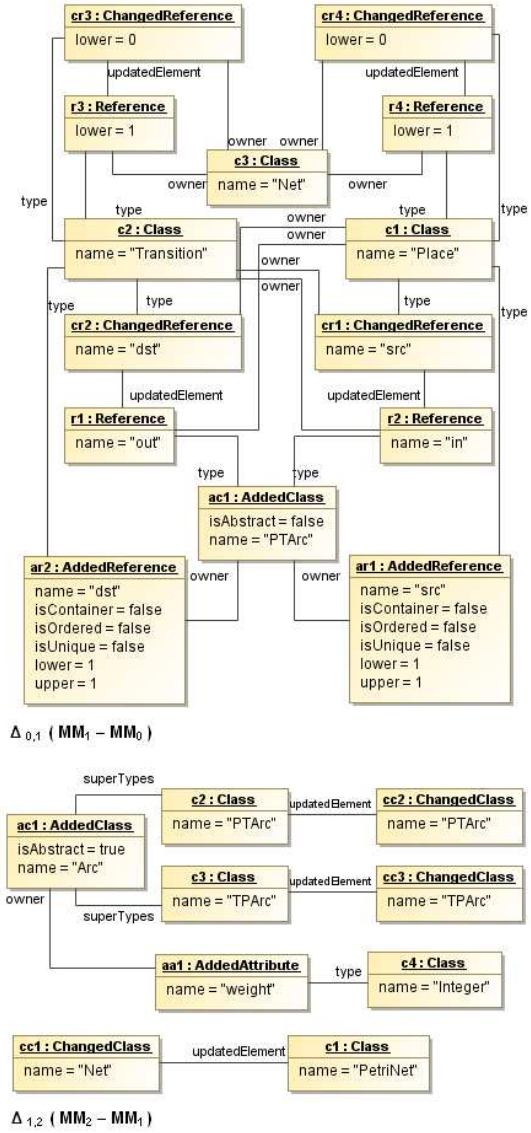

Figure 5. Subsequent Petri Net metamodel adaptations

it to the sequential composition. Otherwise, they are referred to as *parallel dependent* and conflict issues can arise which need to be detected and resolved [5].

Finally, difference documentation can be exploited to reapply changes to arbitrary input models (see [7] for further details) and for managing model co-evolution induced by metamodel manipulations. In the latter case, once differences between metamodel versions have been detected and represented, they have to be partitioned in resolvable and non resolvable scenarios in order to adopt the corresponding resolution strategy. However, this distinction is not always feasible because of parallel dependent changes, i.e. situations where multiple changes are mixed and interdependent one another, like when a resolvable change is in some way related with a non-resolvable one, for instance. In those cases, deltas have to be decomposed in order to isolate the non-resolvable portion from the resolvable one, as illustrated in the next section.

4 Transformational adaptation of models

This section proposes a transformational approach able to consistently adapt existing models with respect to the modifications occurred in the corresponding metamodels. The proposal is based on model transformation and the difference representation techniques presented in the previous section. In particular, given two versions $\texttt{MM}_1$ and $\texttt{MM}_2$ of the same metamodel (see Fig. 6.a), their differences are recorded in a difference model Δ, whose metamodel `KM3Diff` is automatically derived from `KM3` as described in Sect. 3. In realistic cases, the modifications consist of an arbitrary combination of the atomic changes summarized in Tab. 1. Hence, a difference model formalizes all kind of modifications, i.e. non-breaking, breaking resolvable and unresolvable ones. This poses additional difficulties since current approaches (e.g. [23, 11]) do not provide any support to co-adaptation when the modifications are given without explicitly distinguishing among breaking resolvable and unresolvable changes. Our approach consists of the following steps:

i) automatic decomposition of Δ in two disjoint (sub) models, Δ_R and $\Delta_{\neg R}$, which denote breaking resolvable and unresolvable changes;

ii) if Δ_R and $\Delta_{\neg R}$ are *parallel independent* (see previous section) then we separately generate the corresponding co-evolutions;

iii) if Δ_R and $\Delta_{\neg R}$ are *parallel dependent*, they are further refined to identify and isolate the interdependencies causing the interferences.

The distinction between *ii)* and *iii)* is due to fact that when two modifications are not independent their effects depend on the order the changes occur leading to non confluent situations. The confluence can still be obtained by removing those modifications which caused the conflicts as described in Sect. 4.2.

The general approach is outlined in Fig. 6 where dotted and solid arrows represent conformance and transformation relations, respectively, and square boxes are any kind of models, i.e. models, difference models, metamodels, and even transformations. In particular, the decomposition of Δ is given by two model transformations, T_R and $T_{\neg R}$ (right-hand side of Fig. 6.a). Co-evolution actions are directly obtained as model transformations from metamodel changes by means of higher-order transformations, i.e. transformations which produce other transformations [2]. More specifically, the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ (see

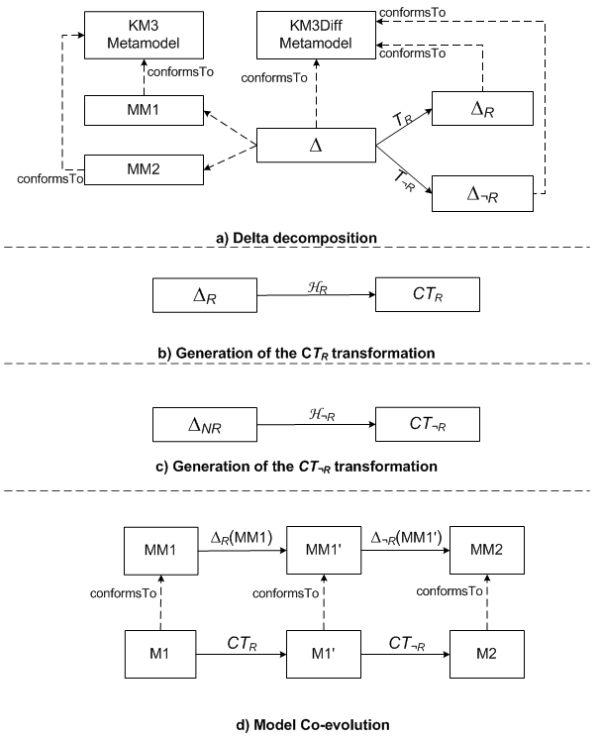

Figure 6. Overall approach

Fig. 6.b and 6.c) take Δ_R and $\Delta_{\neg R}$ and produce the (co-evolving) model transformations CT_R and $CT_{\neg R}$, respectively. Since Δ_R and $\Delta_{\neg R}$ are parallel independent CT_R and $CT_{\neg R}$ can be applied in any order because they operate to disjoint sets of model elements, or in other words

$$(CT_{\neg R} \cdot CT_R)(M_1) = (CT_R \cdot CT_{\neg R})(M_1) = M_2$$

with M_1 and M_2 models conforming to the metamodel $\texttt{MM}_1$ and $\texttt{MM}_2$, respectively (see Fig. 6.d).

The next sections illustrate the approach and its implementation. In particular, we describe the decomposition of Δ and the generation of the co-evolving model-transformations for the case of parallel independent breaking resolvable and unresolvable changes. Finally, in Sect. 4.2 we outline how to remove interdependencies from parallel dependent changes in order to generalize the solution provided in Sect. 4.1. The overall approach has been implemented and the interested reader can download it at [6].

4.1 Parallel independent changes

The generation of the co-evolving model transformations is described in the rest of the section by means of the evolutions the `PetriNet` metamodel has been subject to in Fig. 1. The differences between the subsequent metamodel versions are given in Fig. 5 and have, in turn, to be decomposed to distinguish breaking resolvable and unresolvable modifications.

In particular, the difference $\Delta_{(0,1)}$ from $\texttt{MM}_0$ to $\texttt{MM}_1$ consists of two atomic modifications, i.e. an *Extract metaclass* and a *Restrict metaproperty* change (according to the classification in Tab. 1), which are referring to different sets of model elements. The approach is able to detect parallel independence by verifying that the eventual decomposed differences have an empty intersection. Since *a)* the previous atomic changes are breaking resolvable and unresolvable, and *b)* they do not share any model element, then $\Delta_{(0,1)}$ is decomposed by T_R and $T_{\neg R}$ into the parallel independent $\Delta_{R(0,1)}$ and $\Delta_{\neg R(0,1)}$, respectively. In fact, the former contains the extract metaclass action which affects the elements `Place` and `Transition`, whereas the latter holds the restrict metaproperty changes consisting of the reference modifications in the metaclass `Net`. Analogously, the same decomposition can be operated on $\Delta_{(1,2)}$ (denoting the evolution from $\texttt{MM}_1$ to $\texttt{MM}_2$) to obtain $\Delta_{R(1,2)}$ and $\Delta_{\neg R(1,2)}$ since the denoted modifications do not conflict one another. In fact, the *Rename metaelement* change (represented by `cc1` and `c1` in Fig. 5.b) is applied to `Net`, whereas the *Add obligatory metaproperty* operation involves the new metaclass `Arc` which is supertype of the `PTArc` and `TPArc` metaclasses.

As previously said, once the Δ is decomposed the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ detect the occurred metamodel changes and accordingly generate the co-evolution to adapt the corresponding models. In the current implementation, model transformations are given in ATL, a QVT compliant language part of the AMMA platform [3] which contains a mixture of declarative and imperative constructs. In the Listing 1 a fragment of the $\mathcal{H}_R$ transformation is reported: it consists of a module specification containing a header section (lines 1-2), transformation rules (lines 4-41) and a number of helpers which are used to navigate models and to define complex calculations on them. In particular, the header specifies the source models, the corresponding metamodels, and the target ones. Since the $\mathcal{H}_R$ transformation is higher-order, the target model conforms to the ATL metamodel which essentially specifies the abstract syntax of the transformation language. Moreover, $\mathcal{H}_R$ takes as input the model which represents the metamodel differences conforming to `KM3Diff`.

The helpers and the rules are the constructs used to specify the transformation behaviour. The source pattern of the rules (e.g. lines 15-20) consists of a source type and a OCL [17] guard stating the elements to be matched. Each rule specifies a target pattern (e.g. lines 21-25) which is composed of a set of elements, each of them (as the one at lines 22-25) specifies a target type from the target metamodel (for instance, the type `MatchedRule` from the ATL metamodel) and a set of bindings. A binding refers to a

feature of the type, i.e. an attribute, a reference or an association end, and specifies an expression whose value initializes the feature. $\mathcal{H}_R$ consists of a set of rules each of them devoted to the management of one of the resolvable metamodel changes reported in Tab. 1. For instance, the Listing 1 contains the rules for generating the co-evolution actions corresponding to the *Rename metaelement* and the *Extract metaclass* changes.

```
1 module H_R;
2 create  OUT : ATL  from Delta : KM3Diff;
3 ...
4 rule atlModule {
5  from
6      s: KM3Diff!Metamodel
7  to
8      t : ATL!Module (
9        name <- 'CTR',
10       outModels <- Sequence {tm},
11       inModels <- Sequence {sm},...
12     ),...
13 }
14 rule CreateRenaming {
15   from
16       input : KM3Diff!Class,
17       delta : KM3Diff!ChangedClass
18       ...
19       (not input.isAbstract
20        and input.name <> delta.updatedElement.name...)
21   to
22       matchedRule :  ATL!MatchedRule (
23         name<-input.name + '2' + delta.updatedElement.
               name,
24         ...
25       ),...
26 }
27 rule CreateExtractMetaClass {
28  from
29      cr1: KM3Diff!ChangedReference, cr2: KM3Diff!
           ChangedReference, r1 : KM3Diff!Reference, r2 :
            KM3Diff!Reference, c1 : KM3Diff!Class,
30      c2 : KM3Diff!Class,...
31      ( cr1.updatedElement = r2 and cr1.owner = c2
32        and cr1.type = c1 and ...)
33  to
34      -- MatchedRule generation
35      matchedRule_i_c2 :  ATL!MatchedRule (
36         name<-i_c2.name + '2' + i_c2.name,
37         inPattern <- ip_i_c2,
38         outPattern <- op_i_c2,
39         ...
40      ),...
41 }
42 ...
```

Listing 1. Fragment of the HOT_R transformation

The application of $\mathcal{H}_R$ to the metamodel $\texttt{MM}_0$ in Fig. 1 and the difference model $\Delta_{R(0,1)}$ in Fig. 5 generates the model transformation reported in the Listing 2. In fact, the source pattern of the `CreateExtractMetaClass` rule (lines 28-32 in the Listing 1) matches with the two *Extract metaclass* changes represented in $\Delta_{R(0,1)}$. They consist of the additions of the `PTArc` and `TPArc` metaclasses instead of the direct references between the existing elements `Place` and `Transition`. Consequently, according to the structural features of the involved elements, the `CreateExtractMetaClass` rule generates the transformation $CT_{R(0,1)}$ which is able to co-evolve all the models conforming to $\texttt{MM}_0$ by adapting them with respect to the new metamodel $\texttt{MM}_1$ (see line 1-2 of the Listing 2). In particular, each element of type `Place` has to be modified by changing all the references to elements of type `Transition` with references to new elements of type `PTArc` (see lines 4-23 in the Listing 2). The same modification has to be performed for all the elements of type `Transition` by creating new elements of type `TPArc` which have to be added instead of direct references between `Transition` and `Place` instances (see lines 24-42).

```
1 module CTR;
2 create OUT : MM1 from IN : MM0;
3 ...
4 rule Place2Place {
5   from
6     s : MM1!Place
7     ...
8   to
9     t : MM2!Place (
10      name <- s.name,
11      net <- s.net,
12      out <- s.dst->collect(e |
13        thisModule.createPTArc(e, t)
14      )
15    )
16 }
17 rule createPTArc(s : OclAny, n : OclAny) {
18   to
19     t : MM2!PTArc (
20       src <- s,
21       dst <- n
22     ), ...
23 }
24 rule Transition2Transition {
25   from
26     s : MM1!Transition
27     ...
28   to
29     t : MM2!Transition (
30       net <- s.net,
31       in <- s.dst->collect(e |
32         thisModule.createTPArc(e, t)
33       )
34     )
35 }
36 rule createTPArc(s : OclAny, n : OclAny) {
37   to
38     t : MM2!PTArc (
39       dst <- s,
40       src <- n
41     ), ...
42 }
43 ...
```

Listing 2. Fragment of the generated $CT_{R(0,1)}$ transformation

The management of the breaking and unresolvable modifications is based on the same techniques presented so far for the breaking resolvable case. However, as mentioned in Sect. 2, the involved transformations can not automatically co-adapt the models but are limited to default actions which have to be refined by the designer. Due to space limitation the code of the $\mathcal{H}_{\neg R}$ transformation is not described here. However, the interested reader can download it at [6].

4.2 Parallel dependent changes

As mentioned above, the automatic co-adaptation of models relies on the parallel independence of breaking resolvable and unresolvable modifications, or more formally

$$\Delta_R|\Delta_{\neg R} = \Delta_R; \Delta_{\neg R} + \Delta_{\neg R}; \Delta_R \quad (1)$$

where $+$ denotes the non-deterministic choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. In case the modifications in Tab. 1 refer to the same elements then the order in which such modifications take place matters and does not allow the decomposition of a difference model as, for instance, when evolving $\texttt{MM}_0$ directly to $\texttt{MM}_2$ (although the sub steps $\texttt{MM}_0 - \texttt{MM}_1$ and $\texttt{MM}_1 - \texttt{MM}_2$ are directly manageable as described in the previous section).

A possible approach, which is only sketched in the following, consists in isolating the interdependencies whenever (1) does not hold. The intention is to define an iterative process consisting in *diminishing* the modifications between two metamodels until the corresponding breaking resolvable and unresolvable differences are parallel independent. In particular, let Δ be a difference between two metamodels, then we denote by $\mathcal{P}(\Delta)$ the *difference power-model*, that is the (partially ordered) set of all possible valid sub models of Δ (i.e. fragments of the difference model which are still conforming to the difference metamodel)

$$\mathcal{P}(\Delta) = \{\delta_0 = \phi, \cdots, \delta_i, \delta_{i+1}, \cdots, \delta_n = \Delta\}$$

Then, the solution is the smallest k in $\{0, \cdots, n\}$ such that

$$\Delta^{(k)}; \delta_k = \Delta$$

where $\Delta^{(k)}$ is the difference model between Δ and δ_k, and

$$\Delta^{(k)} = \Delta^{(k)}_R|\Delta^{(k)}_{\neg R}$$

with $\Delta^{(k)}_R$ and $\Delta^{(k)}_{\neg R}$ parallel independent. Hence, the problem of parallel dependence is reduced to the following

$$\Delta = (\Delta^{(k)}_R|\Delta^{(k)}_{\neg R}); \delta_k$$

by applying the higher-order transformation introduced in the previous section. For instance, if we consider ($\texttt{MM}_2 - \texttt{MM}_0$) the solution consists in iteratively finding a difference model which maps $\texttt{MM}_0$ to the intermediate metamodel corresponding to $\texttt{MM}_2$ without the attribute *weight* of the `Arc` metaclass. Therefore, the remaining δ_k in this example is a non resolvable change, while in general it may demand for further iterations of the decomposition process.

5 Related works

Over the last few years, the problem of metamodel evolution and model co-evolution has been investigated by several works [23, 11]. In general, model co-evolution requires the changes to be categorized as *a)* without effects on existing model instances *b)* with simple side effects on models *c)* with side effects demanding for additional management [20]. To this end, the change classification presented in Sect. 2 is inspired by the existing experiences on metamodel evolution. In particular, the work in [23] distinguishes adaptations that ensure instance preservation from manipulations which induce co-evolutions. Metamodel evolutions are specified by QVT relations [16], while co-adaptations are defined in terms of QVT transformations when resolvable changes occur. The main limitations are that co-adapting transformations are not automatically obtained from metamodel modifications and unresolvable changes are not given explicit support. Moreover, using relations instead of difference models does not allow distinguishing metaelement updates from deletion/addition patterns and causes *false-positives* in detecting, for instance, extract metaclass cases. In fact, the only change types which can be precisely caught are the additive and subtractive ones. This problem is (partly) addressed in [11], which advocates for some metamodel difference management by means of *change traces*, although no specific proposal is adopted or given. This work has the merits of classifying changes as breaking/non-breaking and sketching an algorithmic detection of such modifications which is deferred to future work. Similarly to [23], it does not provide any automatic derivation of the co-evolving transformations.

A common aspect to [11, 23] is the atomicity of the changes, i.e. the classified change types are assumed to occur individually, which is far from being a realistic scenario since modifications tend to occur with arbitrary multiplicity and complexity. Additionally, interdependencies may also be present posing severe difficulties in distinguishing the various change types described in Sect. 2.

The solution presented in this paper has a number of similarities with the techniques illustrated in [8], where the authors discuss the possibility to induce model transformations through model weaving. In particular, weaving links are given to establish correspondences between metamodel elements and consequently to derive mappings between corresponding models. If the weaving is seen as a difference representation, the induced transformation can be considered as the automated co-adaptation of existing instances. Nonetheless, the approach in [8] lacks of expressiveness, since only additions and deletions can be represented through the semantics provided by the proposed weaving relationships. As a consequence, the co-adaptation refers only to additive and subtractive cases (as for [23]) and requires the developers to provide explicit support to updative cases.

The issues discussed in this work can be also found in the context of database evolution and metadata handling, which have been demonstrated to share several problems related

to model management [1]. In fact, when schemas evolve to overcome new requirements all the interconnected artefacts need to be co-adapted, like queries, scripts and even existing data. Also in this field, a common solution relies on the separation between schema manipulations causing no or limited updates to existing instances versus modifications requiring deep structural changes and data conversions. Analogously to this paper, simple situations can be automatically supported, while complex ones demands for user intervention, even though the environment can be adequately started-up [10].

6 Conclusions and future work

This paper presented a transformational approach to co-evolution of models which are requested to become conforming to a newer version of their original metamodel. The main points include *1)* starting from metamodel differences the automated generation of co-evolving actions can be obtained by means of higher-order transformations; the adaptation considers both resolvable and non-resolvable changes by providing the designer, in case of knowledge non-determinism, with refinement mechanisms; *2)* the co-adaptation technique deals with the occurrence of multiple change types in the metamodel in order to cope with realistic scenarios; in particular, differences must be decomposed in resolvable and non-resolvable changes.

The previous decomposition can lead to parallel dependent metamodel differences which require an explicit isolation of those modifications which cause the resolvable and non-resolvable changes to be interdependent. By means of the difference powermodel construction given in Sect. 4.2 it is possible to arrange modifications in a lattice which guides the resolvable and non-resolvable differences to be iteratively refined until they become parallel independent.

Apart from the iterative decomposition procedure, the complete approach has been implemented on the AMMA [3] platform and is available for download at [6]. Future work includes the implementation of the powermodel construction the difference refinement depends on. Moreover, a more systematic validation of the approach must necessarily encompass larger population of models and metamodels.

References

[1] P. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Procs of the 1st Conf. on Innovative Data Systems Research (CIDR)*, 2003.

[2] J. Bézivin. On the Unification Power of Models. *Jour. on Software and Systems Modeling (SoSyM)*, 4(2):171–188, 2005.

[3] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *European MDAFA Workshops*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.

[4] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[5] A. Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, University of L'Aquila, Computer Science Dept., 2008.

[6] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Implementation of an automated co-evolution of models through atl higher-order transformations. http://www.di.univaq.it/diruscio/CoevImpl.php, 2008.

[7] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.

[8] M. D. Del Fabro and P. Valduriez. Semi-automatic Model Integration using Matching Transformations and Weaving Models. In *The 22th ACM SAC - MT Track*, pages 963–970. ACM, 2007.

[9] J.-M. Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Procs. of the Int. Workshop ELISA at ICSM*, September 2003.

[10] R. Galante, N. Edelweiss, and C. dos Santos. Change Management for a Temporal Versioned Object-Oriented Database. In *Procs. of the 21st ER*, volume 2503 of *LNCS*, pages 1–12. Springer, 2002.

[11] B. Gruschko, D. Kolovos, and R. Paige. Towards Synchronizing Models with Evolving Metamodels. In *Procs of the Work. MODSE*, 2007.

[12] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[13] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Work. MDSD*, 2004.

[14] T. Mens, J. Buckley, A. Rashid, and M. Zenger. Towards a taxonomy of software evolution. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel, 2003.

[15] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04*. http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[16] Object Management Group (OMG). MOF QVT Final Adopted Specification, 2005. OMG Adopted Specification ptc/05-11-01.

[17] Object Management Group (OMG). OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[18] J. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *Procs of the 46th Int. Conf. TOOLS EUROPE 2008*, 2008. To appear.

[19] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[20] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004.

[21] A. Toulmé. The EMF Compare Utility. http://www.eclipse.org/modeling/emft/.

[22] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *Proceedings of ESEC/FSE*, pages 295–304, New York, NY, USA, 2007. ACM.

[23] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In E. Ernst, editor, *Proceedings of the 21st ECOOP*, volume 4069 of *LNCS*. Springer-Verlag, July 2007.