Managing dependent changes in coupled evolution*

Antonio Cicchetti[1], Davide Di Ruscio[2], and Alfonso Pierantonio[2]

[1] School of Innovation, Design and Engineering
Mälardalen University,
SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se

[2] Dipartimento di Informatica
Università degli Studi dell'Aquila
Via Vetoio, Coppito I-67010, L'Aquila, Italy
{diruscio|alfonso}@di.univaq.it

Abstract. In Model-Driven Engineering models and metamodels are not preserved from the evolutionary pressure which inevitably affects almost any artefacts. Moreover, the coupling between models and metamodels implies that when a metamodel undergoes a modification, the conforming models require to be accordingly co-adapted. One of the main obstacles to the complete automation of the adaptation process is represented by the dependencies which occur among the different kinds of modifications. The paper illustrates a dependency analysis, classifies such dependencies, and proposes a metamodeling language driven resolution which is independent from the evolving metamodel and its underlying semantics. The resolution enables a decomposition and consequent scheduling of the adaptation steps allowing the full automation of the process.

1 Introduction

Model Driven Engineering (MDE) [1] is increasingly gaining acceptance as a mean to leverage abstraction and render business logic resilient to technological changes. Coordinated collections of models and modelling languages are used to describe software systems on different abstraction layers and from different perspectives [2]. 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.

In a model-centric vision of software-development, models and metamodels are not preserved from the evolutionary pressure which inevitably affects almost any artefacts involved in the process [3]. Moreover, the coupling between models and metamodels implies that when a metamodel undergoes a modification, the conforming models require to be accordingly *co-adapted*[3] not to let them become invalid. This adaptation process is difficult, error-prone and can give place to inconsistencies between the

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

[3] The terms co-adaptation, co-evolution, and coupled evolution will be at some extent used as synonyms throughout the paper.

metamodel and the related artefacts, if not supported by any automation. Such an issue becomes even more 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 [4]. The management of coupled evolution is intrinsically complex and requires the capability of *a) differencing*, i.e. determining the differences between two versions of the same metamodel and *b) adaptation*, that is a transformational process able to partly or fully automatize the adaptation of the models according to the modifications detected in the previous step. Recently, these aspects have been investigated by several works, while some focused on the problem of metamodel matching (e.g., [5]), most of them concentrated on the adaptation by either assuming that change traces, for instance, are somehow available or addressing only atomic modifications (e.g., [4, 6, 7]), see Sect. 2.1 for a detailed discussion. Unfortunately, supposing the availability of predefined information about changes and assuming only atomic operations is not always practicable, because metamodels usually evolve in a complex way without keeping track of the applied changes.

This paper proposes a transformational approach to co-adaptation which is agnostic of the differencing method and considers complex modifications of metamodels, in contrast with current approaches [4, 6, 7]. As shown in [8], the adaptation is defined as the parallel composition of two different transformations which are automatically derived from the *breaking resolvable*, and *breaking unresolvable* changes. Unfortunately, the occurrence of dependencies between these two kind of changes compromises the parallel independence of the generated transformations, and thus the complete automation of the co-adaptation. This work enhances the work in [8] by proposing a dependency analysis which underpins a resolution strategy allowing the correct scheduling of the adaptation steps. All the metamodel change dependencies have been considered and for each of them a resolution schema is proposed enabling the complete automation of the adaptation. Interestingly, the technique is independent from the metamodel and its underlying semantics, since it relies only on the definition of the metamodeling language.

The structure of the paper is as follows. In Sect. 2 a discussion about the related work and the background is presented. Next section analyzes the metamodel change dependencies and discusses the countermeasures to adopt in order to resolve them. Finally some conclusions are drawn.

2 Metamodel evolution and model co-evolution

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. A possible solution is the adoption of mechanisms of model co-evolution, i.e. models are migrated in new instances according to the changes of the corresponding metamodel. In the following, related works are illustrated to give an overall view of the problem, current solutions, and the issues which are still open.

2.1 Related work

The problem of co-evolution presents intrinsic difficulties. In [7] the authors introduce a new language, COPE, to support the adaptation of models with respect to metamodel updates. However, the language is mainly exploited to provide helpers in instance co-adaptations and not to introduce a generative approach based on metamodel variations. In [4, 6, 9] the authors try to improve the degree of automation, by considering all the possibile metamodel manipulations and distinguishing them with respect to the effects they have on the existing instances. In particular, metamodel changes are classified in *(i) non-breaking changes* that do not break the conformance of models once the corresponding metamodel has been modified, *(ii) breaking and resolvable changes* which break the conformance of models even though they can be automatically co-adapted, and *(iii) breaking and unresolvable changes* that break the conformance of models which can not be automatically co-evolved and user intervention is required. Such a categorization suggests to support model co-evolution by separating the various forms of metamodel revisions and then by adopting the appropriate countermeasures. For instance, in [4] metamodel evolutions are specified by QVT relations, 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. This problem is (partly) addressed in [6], which advocates for some metamodel difference management by means of *change traces*, although no specific proposal is adopted or given.

In [10] the authors discuss the possibility to induce model transformations through model weaving. In particular, weaving links are given to establish correspondences (or matchings) 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 [10] lacks of expressiveness, since only additions and deletions can be represented through the semantics provided by the proposed weaving relationships. The problem of *metamodel matching* is also discussed in [5] where techniques based on schema matching algorithms are used to compute metamodel alignments.

The co-evolution problem is also investigated in the context of database evolution and metadata handling, which have been demonstrated to share several problems related to model management [11]. 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 model co-evolution, simple situations can be automatically supported, while complex ones demand for user intervention, even though the environment can be adequately started-up [12].

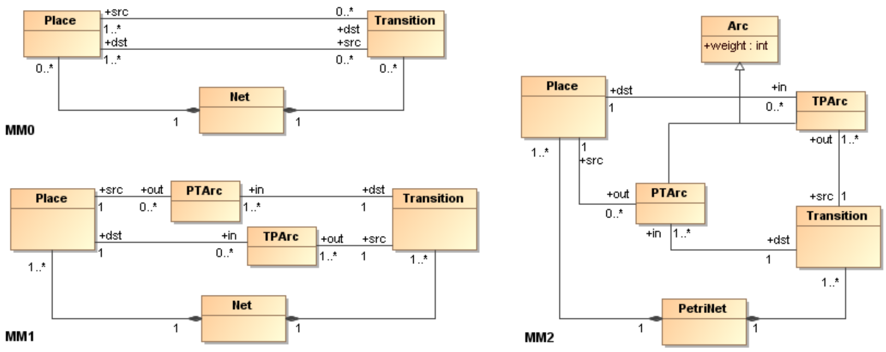


Fig. 1. Petri Net metamodel evolution

2.2 Supporting complex metamodel changes

A common aspect that seems to underlay current approaches to co-evolution is the atomicity of the changes, i.e. the classified change types are assumed to occur individually, which is not always the case 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. To clarify such problems the sample evolution of the (simplified) Petri Net metamodel depicted in Figure 1 will be considered in the rest of the section. In particular, the initial metamodel $\mathtt{MM}_0$ consists of `Place`s and `Transition`s, 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 $\mathtt{MM}_1$, 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, thus allowing to add further properties to relationships between places and transitions. Since `PTArc` and `TPArc` both represent arcs, they have been generalized in $\mathtt{MM}_2$ by the new abstract class `Arc` encompassing the integer metaproperty `weight`. Finally, the metaclass `Net` has been renamed into `PetriNet`.

The modifications applied to the Petri Net metamodel $\mathtt{MM}_0$ to obtain $\mathtt{MM}_1$ consists of breaking and resolvable changes. In fact the addition of the new `PTArc` and `TPArc` metaclasses breaks the conformance of the existing models to $\mathtt{MM}_0$ since, according to the new metamodel $\mathtt{MM}_1$, `Place` and `Transition` instances have to be related through `PTArc` and `TPArc` elements. However, models can be automatically migrated by adding for each couple of `Place` and `Transition` entities two additional `PTArc` and `TPArc` instances between them. An automatic model adaptation cannot be performed when $\mathtt{MM}_1$ is changed to get $\mathtt{MM}_2$ because of the breaking and unresolvable modifications. In particular, in this case, 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.

All the scenarios of model co-adaptations, like the one of the Petri Net example, can be managed with respect to the possible metamodel modifications which can be distinguished into *additive*, *subtractive*, and *updative* [8]. By going into more details, with additive changes we refer to the following metamodel element additions:

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

Table 1. Changes classification

- *Add metaclass or metaproperty*, introducing new metaclasses or metaproperties is a common practice in metamodel evolution which gives place to metamodel extensions;
- *Generalize metaproperty*, a metaproperty is generalized when its multiplicity or type are relaxed, for instance the cardinality is modified from `3..n` to `0..n`, or a type is substituted with its supertype;
- *Pull metaproperty*, a metaproperty `p` is pulled in a superclass `A` and the old one is removed from a subclass `B`;
- *Extract superclass*, a superclass is extracted in a hierarchy and a set of properties is pulled on.

Subtractive changes consist of the deletion of some of the existing metamodel elements:

- *Eliminate metaclass*, a metaclass is deleted by giving place to a sub metamodel of the initial one;
- *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`;
- *Flatten hierarchy*, to flatten a hierarchy means eliminating a superclass and introducing all its properties into the subclasses;
- *Restrict metaproperty*, a metaproperty is restricted when its multiplicity or type are enforced, for example the cardinality is modified from `0..*` to `0..10`, or a type is substituted with one of its subtypes.

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

- *Change metaproperty type*, the type of a metaproperty is updated and the new type has not particular relationships with the old one;
- *Rename metaelement*, a metaelement is renamed;
- *Move metaproperty*, it consists of moving a property `p` from a metaclass `A` to a metaclass `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.

Such classification plays a key role in a transformational approach to model co-evolution presented by the authors in [8] and illustrated in Figure 2. The implementation of the approach relies on the KM3 metamodeling language [13] even though it is general and can be applied to any other meta-metamodel like OMG/MOF [14] or EMF/Ecore [15]. In particular, given two versions $\mathtt{MM}_1$ and $\mathtt{MM}_2$ of the same metamodel, their differences are recorded in a difference model Δ, whose metamodel `KM3Diff` is automatically derived from `KM3` and shown in Figure 3. Essentially, for each metaclass `MC` of the KM3 metamodel, the additional metaclasses `AddedMC`, `DeletedMC`, and `ChangedMC` are generated in order to represent additions, deletions, or changes, respectively, of `MC` instances [16].

In realistic cases, the metamodel modifications represented in the model Δ consist of anarbitrary combination of the atomic changes in Tab. 1. Hence, a difference model formalizes all kind of modifications, i.e. non-breaking, breaking resolvable and unresolvable ones. In this respect, the adopted difference representation approach is crucial. In particular, if the representation of the updates is too gross-grained, then the co-adaptation acts with less efficacy. For instance, if the introduction of `PTArc` and `TPArc` in the sample $\mathtt{MM}_0$ would be represented as the deletion of the current associations and the addition of those new entities, all the existing connections between arcs and transitions would be lost in the co-adaptation process. In fact, `PTArc` and `TPArc` would be interpreted as new relationships between arcs and transitions instead of being a refinement of them. Once the metamodel changes have been calculated (as for instance in [5]) and represented in Δ, such a difference model is automatically decomposed in two disjoint (sub) models, Δ_R and $\Delta_{\neg R}$ [8], which denote breaking resolvable and unresolvable changes, respectively. The decomposition is given by two model transformations, T_R and $T_{\neg R}$ (see Figure 2.a).

As previously said, the possibility to have a set of dependencies among the several parts of the evolution makes the updates not always distinguishable as single atomic steps of the metamodel revision. In such situations, the modifications in Tab. 1 refer to the same elements and then the order in which such modifications take place matters and does not allow the decomposition of a difference model in Δ_R and $\Delta_{\neg R}$, like for instance when evolving $\mathtt{MM}_0$ directly to $\mathtt{MM}_2$ in Figure 1 (although the sub steps $\mathtt{MM}_0 - \mathtt{MM}_1$ and $\mathtt{MM}_1 - \mathtt{MM}_2$ are directly manageable). In these cases Δ_R and $\Delta_{\neg R}$ are said *parallel dependent* and they have to be further refined to identify and isolate the interdependencies causing the interferences. If Δ_R and $\Delta_{\neg R}$ are *parallel independent* then corresponding co-evolutions are generated separately. In particular, co-evolution actions are directly obtained as model transformations from the calculated metamodel changes by means of higher-order transformations, i.e. transformations which produce other transformations [17]. More specifically, two different higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ take Δ_R and $\Delta_{\neg R}$ and produce the (co-evolving) model transformations CT_R

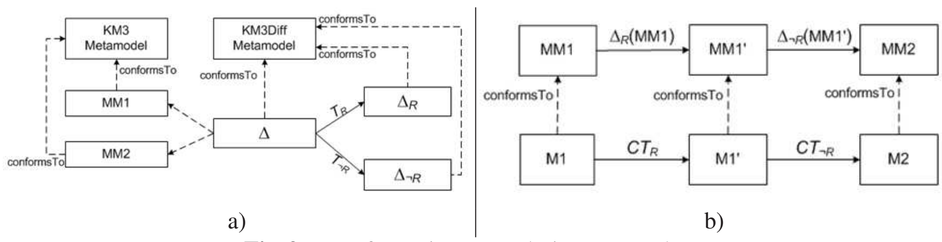


Fig. 2. Transformative co-evolution approach

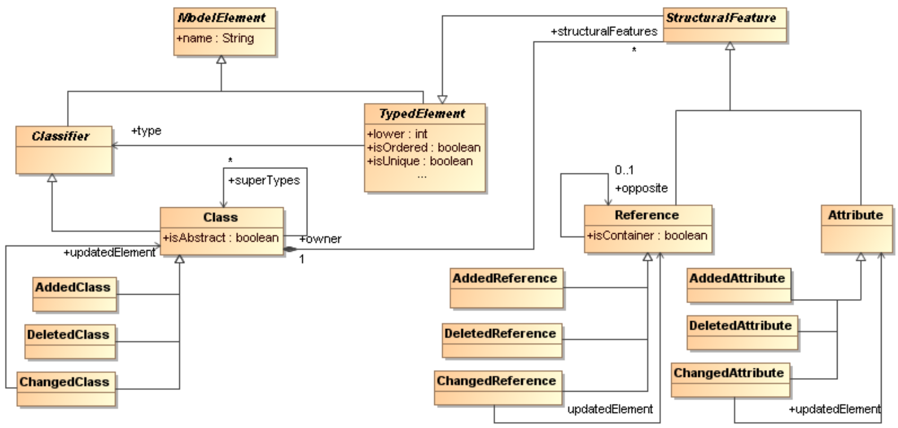


Fig. 3. Fragment of the generated difference KM3 metamodel

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 (see Figure 2.b). The main problem in having such kind of interdependencies is in the *nondeterminism* given by the following

$$\Delta_R | \Delta_{\neg R} \neq \Delta_R ; \Delta_{\neg R} + \Delta_{\neg R} ; \Delta_R$$

denoting with $+$ the nondeterministic choice. In the next section, we proposes a dependency analysis and resolution criteria to decompose and schedule the modifications in order to resolve the dependencies according to a comprehensive classification of them as they can occur in a metamodel evolution.

3 Dealing with parallel dependent changes

The automatic co-adaptation approach recalled in the previous section relies on the parallel independence of breaking resolvable and unresolvable modifications. For instance, when evolving the sample PetriNet metamodel $\mathtt{MM}_0$ in Figure 1 directly to $\mathtt{MM}_2$, the approach cannot be directly applied unless the dependent changes in Δ_R and $\Delta_{\neg R}$ are identified and resolved. In particular, in the example, the *Add obligatory metaclass* modification, consisting of the addition of the attribute `weight` in the metclass *Arc*, depends on the addition of this new metaclass induced by the *Extract abstract metaclass* change. Such a dependence is due to the reference `owner` which, according to the KM3 metamodel, needs to be specified for each structural feature.

Being more precise, given two versions of a same metamodel and a model Δ which represents their differences, the models Δ_R and $\Delta_{\neg R}$ obtained from the decomposition of Δ to isolate breaking resolvable and unresolvable modifications, respectively, are parallel dependent when the source and the target elements of the following references (defined in the KM3 difference metamodel) are not in the same difference model:

– $owner : StructuralFeature \rightarrow \{AddedClass, ChangedClass\}$, all the attributes and references defined in a given metamodel are related to a corresponding

Unresolvable changes	Resolvable changes: (1) Extract (abstract) superclass	(2) Push metaproperty	(3) Move metaproperty	(4) Extract/inline metaclass
(A) Add obligatory metaclass	R,¬R (superTypes)	¬R (owner)	¬R (owner)	-
(B) Add obligatory metaproperty	R (owner,type)	-	-	R (owner,type)
(C) Pull metaproperty	R (owner)	-	-	-
(D) Extract (non abstract) superclass	R,¬R (superTypes)	-	¬R (owner)	R,¬R (superTypes)
(E) Change metaproperty type	R (type)	-	-	R (type)

Table 2. Metamodel change dependencies

class which represents their owner. If a given structural feature *sf* belongs to Δ_R (or $\Delta_{\neg R}$) and its owner metaclass *mc* to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In this case, *owner(sf)* can be specified once *mc* has been added or modified;

– $type : TypedElement \rightarrow \{AddedClass, ChangedClass\}$, given an element *te*, *type(te)* refers to the added or modified classifier *mc* which represents its type. In this respect, if a typed element *te* belongs to Δ_R (or $\Delta_{\neg R}$) and its type *mc* to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In this case, *type(te)* can be specified once *mc* has been added or modified;

– $superTypes : Class \rightarrow \{AddedClass, ChangedClass\}^*$, in order to specify hierarchies of classes, the *superTypes* reference is available to define all the superclasses c_i of a given class c. If a given class c belongs to Δ_R (or $\Delta_{\neg R}$) and its superclasses c_i to $\Delta_{\neg R}$ (or Δ_R), then a parallel dependence occurs. In fact *superTypes(c)* can be specified once the superclasses c_i have been added or modified.

Because of such references, many of the metamodel changes recalled in the previous section may give place to parallel dependencies which are summarized in Table 2. In particular, the rows of the table reports unresolvable changes whereas the resolvable ones are given in the columns. Non empty cells represent the dependencies which may occur because of the corresponding couple of unresolvable and resolvable changes which might interfere one with another because of the specified reference. For instance, the cell `B1` is not empty since an *Add obligatory metaproperty* modification and an *Extract abstract superclass* one may give place to a dependence because of the references *owner* or *type*. In particular, an added obligatory metaproperty may have as owner or type the new superclass obtained by means of an *Extract abstract superclass* modification. In this respect, as in the PetriNet example, the dependence can be sorted out by applying the resolvable change before the unresolvable one (this is the meaning of the `R` in the cell `B1`).

The rest of the section is organized as follows: all the metamodel change dependencies summarized in Table 2 will be described in Section 3.1. The identification and the resolution of the dependencies occurring in large difference models are discussed in Section 3.2.

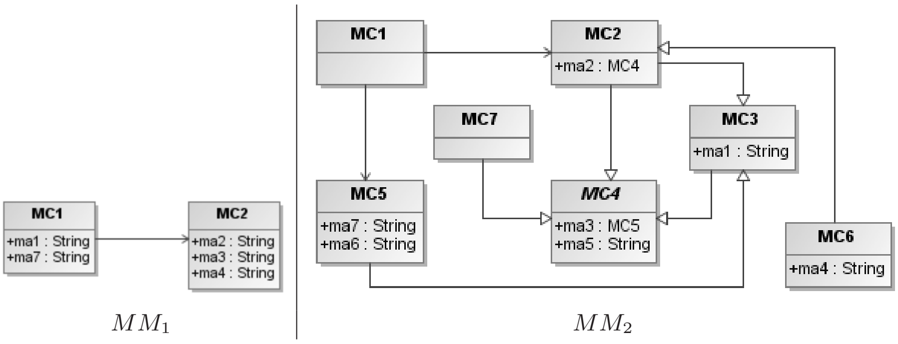


Fig. 4. Sample metamodel evolution

3.1 Classification of change dependencies

The description of the parallel dependent changes summarized in Table 2 exploits the sample metamodel evolution reported in Figure 4. The differences between the sample metamodels MM_1 and MM_2 are represented in the difference model in Figure 5 which has been decomposed in the corresponding Δ_R and $\Delta_{\neg R}$ in Figure 6.

A1. Both the *Add obligatory metaclass* and *Extract abstract superclass* modifications give place to new metaclasses. Such modifications are parallel dependent if the metaclass added by the former is subclass of the metaclass added by the latter (or viceversa). For instance, in the running example, an *Add obligatory metaclass* modification has been executed to add the new metclass `MC7` as specialization of `MC4` which is a new abstract metaclass that has been added as superclass of the existing `MC2`. The addition of `MC7` is represented by the element `ac3` in the model $\Delta_{\neg R}$ whereas the addition of the metaclass `MC4` is represented in the Δ_R by means of the element `ac2`. Such modifications are parallel dependent since `supertTypes` of the added `MC7` refers to the metaclass `MC4` whose addition is in Δ_R.

B1. The owner or the type of a new attribute obtained by means of an *Add obligatory metaproperty* modification may be a new class which has been added by means of an *Extract abstract superclass* operation. For instance, in the running example the new meta attribute `ma5` has been added as represented by the element `aa1` in $\Delta_{\neg R}$ and its `owner` refers to the metaclass `MC4` which has been obtained through the *Extract abstract superclass* modification previously described.

C1. The *Pull metaproperty* modification moves a metaproperty `p` from a subclass `B` to the superclass `A`. If such superclass is obtained through an *Extract abstract superclass* modification, a parallel dependence occur since in order to set the reference `owner` of `p`, the metaclass `A` has to be added first. For instance, the metaproperty `ma3` has been moved from `MC2` to the new metaclass `MC4` by means of a *Pull metaproperty* modification (see the elements `ac2` and `a1` in $\Delta_{\neg R}$) Such modification depends on the addition of the metaclass `MC4` which is represented in Δ_R as described above.

D1. The *Extract non abstract superclass* modification extracts a non abstract superclass `A` in a hierarchy. If `A` is superclass of an abstract class obtained after an *Extract abstract superclass* modification (or viceversa), a parallel dependence is raised because of the `superTypes` reference. For instance, an *Extract non abstract superclass* modification

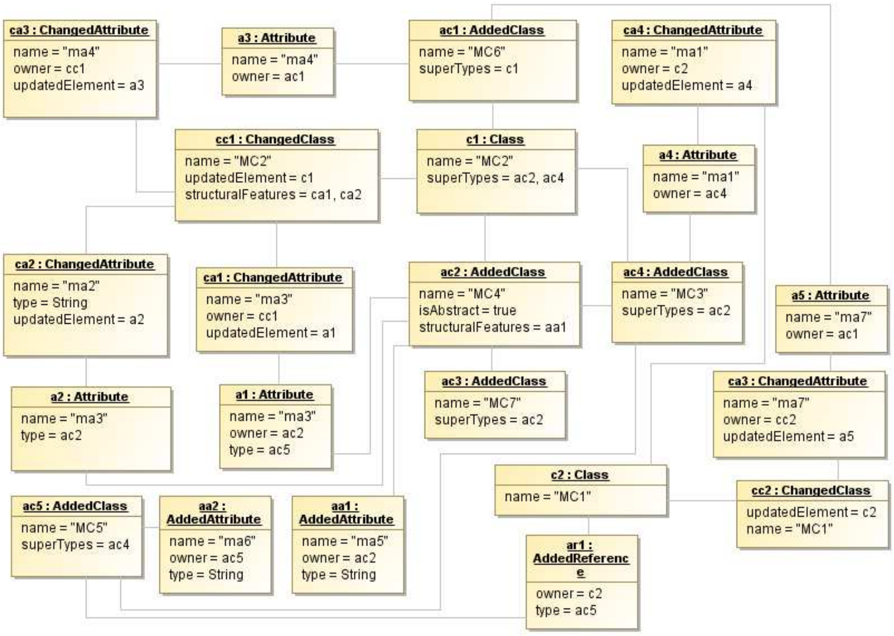
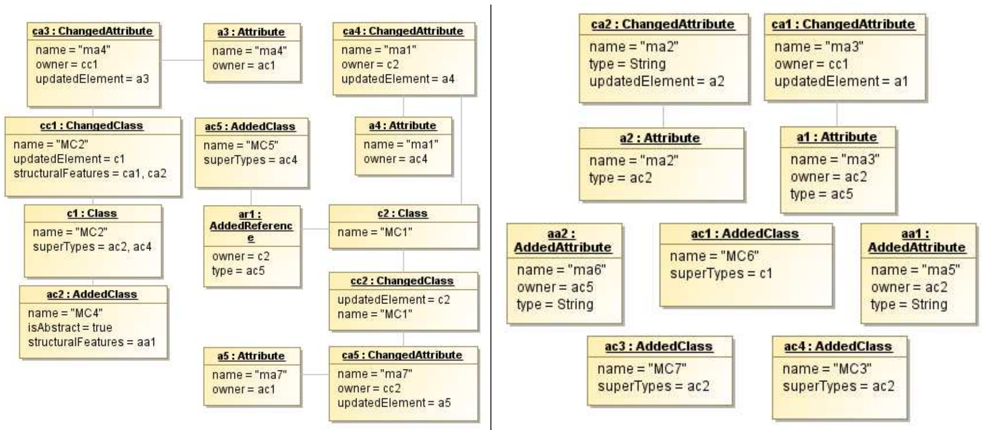


Fig. 5. Representation of the sample metamodel modifications

has been performed to create the new metaclass `MC3` as superclass of `MC2` (see the element `ac4` in $\Delta_{\neg R}$). In this case, the dependence `D1` occurs since `MC3` also specializes the metaclass `MC4` (see the reference `superTypes` of the element `ac4` in $\Delta_{\neg R}$ to the element `ac2` in Δ_R) which has been obtained after an *Extract abstract superclass* modification represented in Δ_R.

E1. If the type of a metaproperty is changed to the abstract class obtained by means of an *Extract non abstract superclass* modification, a parallel dependence occurs because of the `type` reference. For instance, the type of the attribute `ma2` in the metaclass `MC2` has been changed from `String` to `MC4`. This is a *Change metaproperty type* modification and is represented in $\Delta_{\neg R}$ by means of the elements `ca2` and `a2`. However, since the new type of the attribute `ma2` is a class obtained by means of an *Extract abstract superclass* modification, the dependence *E1* occurs.

A2. The *Push metaproperty* modification deletes a metaproperty `p` from a superclass `A` and clones it in all the subclasses `C` of `A`. If the subclasses `C` have been added by means of *Add obligatory metaclass* modifications, parallel dependencies occur because of the `owner` reference. In the running example, an *Add obligatory metaclass* change has been performed to add `MC6` as specialization of `MC2`. Such a modification is represented in $\Delta_{\neg R}$ by means of the element `ac1`. Moreover, a *Push metaproperty* change has been executed to change the owner of the attribute `ma4` from the metaclass `MC2` to the just added `MC6`. This modification is represented in Δ_R by the elements `ca3` and `a3` instances of the metaclasses `ChangedAttribute`, and `Attribute`, respectively. The addition of `MC6` and the owner change of the attribute `ma4` are an example of the dependence *A2*.



a. Resolvable changes (Δ_R) b. Unresolvable changes ($\Delta_{\neg R}$)

Fig. 6. Decomposed difference model

A3. Similarly to the dependence *A3*, *A2* occurs because of the reference `owner` when a metaproperty is moved to a metaclass added by means of an *Add obligatory metaclass* modification. For instance, the attribute `ma7` has been moved from the metaclass `MC1` to the new metaclass `MC6` by means of the *Move metaproperty* change represented in Δ_R by the elements `ca5` and `a5`. Such a modification depends on the *Add obligatory metaclass* change which has to be performed first in order to add the metaclass `MC6` and update the value of the reference `owner` of the attribute `ma7`.

D3. A metaproperty can be moved to a new metaclass obtained by means of an *Extract non abstract superclass* modification. In this case, because of the `owner` reference, a dependence occurs and to set the owner of the moved property, the new non abstract metaclass has to be extracted first. In the running example, a *Move metaproperty* modification has been executed to move the attribute `ma1` from the metaclass `MC1` to `MC3` as represented by the elements `ca4` and `a4` in Δ_R. However, since the new owner of the attribute `ma1` is the metaclass `MC3` (obtained through an *Extract non abstract superclass* represented in $\Delta_{\neg R}$) the dependence *D3* takes place.

B4. The *Extract metaclass* operation means to create a new metaclass and move the relevant fields from the old metaclass to the new one and relate them. For instance, in Figure 4 an *Extract metaclass* operation has been performed to create the new metaclass `MC5` associated with the existing `MC1` (see the elements `cc2`, `c2`, `ar1`, and `ac5` in Δ_R). Consequently, if a new metaproperty `mp` is created by means of an *Add obligatory metaproperty* modification, a dependence with the *Extract metaclass* modification can be raised if the `type` or the `owner` of `mp` is the extracted metaclass. For instance, the new attribute `ma6` has been added in `MC5` as represented by the element `aa2` in $\Delta_{\neg R}$, and the modification depends on the *Extract metaclass* operation since the owner of the new attribute `ma6` is the extracted metaclass `MC5`.

D4. As previously said, the *Extract non abstract superclass* modification extracts a non abstract superclass `A` in a hierarchy. If `A` is superclass of a class obtained by means of an *Extract metaclass* modification (or viceversa), a parallel dependence is raised because of the `superTypes` reference. For instance, the metaclass `MC5`, obtained through an

Extract metaclass modification, has been added as specialization of the class `MC3` which has been created by means of *Extract non abstract superclass* change giving place to dependent modifications.

E4. An existing metaproperty can be modified by setting its type to a metaclass which has been added by means of an *Extract metaclass* modification. In this case, dependent modifications have been performed which need to be sorted out. For instance, the type of the attribute `ma3` moved to the new metaclass `MC4` has been changed from `String` to the new metaclass `MC5` by means of a *Change mataproperty type* operation represented by the elements `ca1` and `a2` in $\Delta_{\neg R}$. Since the new type of the attribute is a class obtained by means the *Extract metaclass* modification, the dependence *E4* takes place.

When the evolution of a metamodel consists of complex modifications, the decomposition in resolvable and unresolvable changes can easily give place to dependencies which are usually difficult to be identified and sorted out by hand. In the next section we propose a formal approach to support the identification and the resolution of such dependencies.

3.2 Identification and resolution of change dependencies

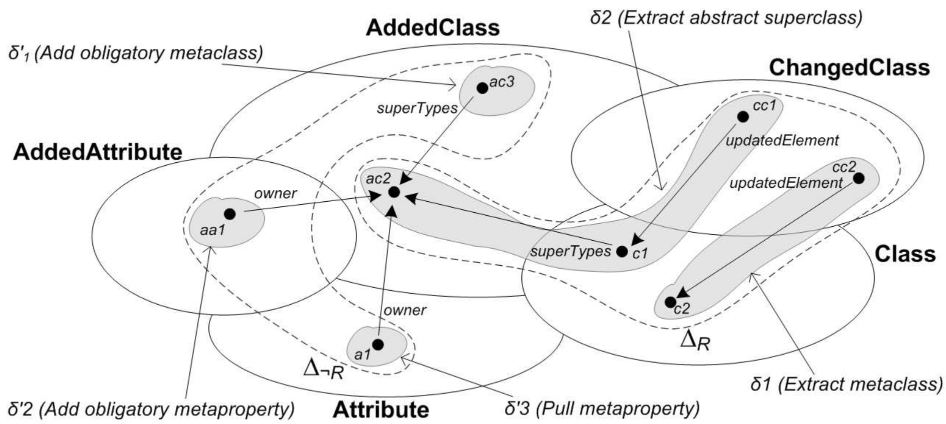
In this section we propose an approach to identify and resolve the dependencies which have been discussed in the previous section. To this end, an algebraic encoding of the difference models is considered in order to automatically sort out the dependent changes.

In particular, an algebra signature is directly derived from the KM3 difference metamodel whose elements define sorts and functions as reported in Figure 7, this operation can be performed in an automated way as shown in [18, 19]. In other words, the metamodel induces the signature Σ composed of sorts (S) and functions (OP): for each non abstract metaclass of the metamodel a correspondent set in S is defined, and the functions in OP are induced by the attributes and references of all the metaclasses. For instance, the attribute `name` of the metaclass `Class` induces the definition of the function *name: Class → Boolean*. Moreover, to specify the `type` of an `Attribute`, the function *type: Attribute → Class* is defined with respect to the property *type* of the abstract metaclass `TypedElement` which is superclass of the `Attribute` one.

The sets and the functions in Figure 7 enables the encoding of models conforming to the KM3 difference metamodel as in the example in Figure 8 which depicts the

$$
\begin{aligned}
&\Sigma = (S, OP)\\
&S := \{Class, AddedClass, ChangedClass, Attribute,\\
&\quad AddedAttribute, ChangedAttribute, \ldots\}\\
&OP :=\\
&\quad name : Class \to String\\
&\quad name : Attribute \to String\\
&\quad isAbstract : Class \to Boolean\\
&\quad isPrimary : Attribute \to Bool\\
&\quad type : Attribute \to Class\\
&\quad owner : Attribute \to Class\\
&\quad \ldots
\end{aligned}
$$

Fig. 7. Fragment of the signature induced by the KM3 difference metamodel



name(a1) = "ma3"	name(c2) = "MC1"	name(ac3) = "MC7"
name(aa1) = "ma5"	name(cc1) = "MC2"	name(cc2) = "MC1"
name(ac2) = "MC4"	name(c1) = "MC2"	...

Fig. 8. Sample difference model encoding

encoding of a fragment of the difference models in Figure 6. The resolvable and the unresolvable modifications are also distinguished (see the dashed parts which enclose Δ_R and $\Delta_{\neg_R}$, respectively) and each of them consists of a set of the atomic metamodel changes described in the previous section. For instance, the modification δ_2 in Δ_R corresponds to the *Extract abstract superclass* modification which has been applied to the metamodel MM_1 in Figure 4 to add the metaclass `MC4` in MM_2. Moreover, the *Add obligatory metaclass* modification which has been executed to add the metaclass `MC7` has been represented by δ'_1 in $\Delta_{\neg R}$. As discussed in the previous section, the latter modification depends on the former according to the case `A1` in Table 2. Such a dependence can be noticed also by considering the encoding in Figure 8. In fact, the reference `superTypes` of the elements `ac3` in $\Delta_{\neg R}$ has `ac2` as value which is in Δ_R. In this respect, the modification δ'_1 depends on δ_2, hence $\Delta_{\neg R}$ depends on Δ_R.
Being more formal, by considering the *owner*, *superTypes*, and *type* functions defined at the beginning of the section, the following definitions can be given:

Definition 1. *Let* $\delta_1 = \{a_1, a_2, \ldots, a_n\}$ *and* $\delta_2 = \{b_1, b_2, \ldots, b_m\}$ *be two atomic metamodel changes.* δ_1 depends on δ_2 *if exists a couple* (a_i, b_j)*,* $i \in \{1 \ldots n\}$*,* $j \in \{1 \ldots m\}$*, such that* $owner(a_i) = b_j$ *or* $type(a_i) = b_j$ *or* $superTypes(a_i) = b_j$*.*

Definition 2. *Let* $\Delta_1 = \{\delta_1, \delta_2, \ldots, \delta_n\}$ *and* $\Delta_2 = \{\delta'_1, \delta'_2, \ldots, \delta'_m\}$ *be two difference models,* Δ_1 depends on Δ_2 *if exist a couple* (δ_i, δ'_j)*,* $i \in \{1 \ldots n\}$*,* $j \in \{1 \ldots m\}$*, such that* δ_i depends on δ'_j*.*

It is important to stress how the functions above are part of the KM3 definition and are the only responsible for the dependencies among the breaking resolvable and breaking unresolvable changes. As a consequence, this makes the technique independent from the metamodel and its underlying semantics.

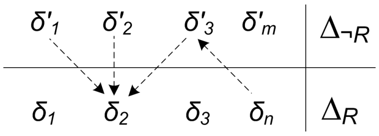


Fig. 9. Fragment of the sample change dependencies

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

$$\Delta_R | \Delta_{\neg R} = \Delta_R ; \Delta_{\neg R} + \Delta_{\neg R} ; \Delta_R \tag{1}$$

where $+$ denotes the non-deterministic choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. If change dependencies are identified they have to be sorted out in order to recover the parallel independence condition. In this respect, according to Table 2, the discovered dependencies induce the order in which changes have to be applied. For instance, Figure 9 contains a fragment of the sample metamodel changes presented above with their dependencies depicted by means of dashed arrows. By taking into account such dependencies and the resolution criteria presented above, the correct scheduling of modifications is as follows

$$(\Delta_R - \{\delta_n\}) \mid (\Delta_{\neg R} - \{\delta'_1, \delta'_2, \delta'_3\}) \;\; ; \;\; \{\delta'_1 \mid \delta'_2 \mid \delta'_3\} \;\; ; \;\; \delta_n \tag{2}$$

The identification of change dependencies can be easily automatized by translating each non-empty entry in Table 2 into first-order logic predicates and thus implemented in OCL [20], for instance.

Finally, it is worth to mention that cyclic change dependencies cannot occur. In particular, because of the typing of the functions *type*, *owner*, and *superTypes* the only admitted cycle might be caused by the last one since it has the set *Class* as domain and codomain. However, having a cyclic dependence because of such a function would give the possibility to define cyclic hierarchies which are not admitted in general.

4 Conclusions and future work

In this paper, we have presented an approach that automatizes the adaptation of models whenever the corresponding metamodel is subject to evolution, i.e., to arbitrary, complex and, possibly *non-monotonic* modifications. To the best of our knowledge, the existing approaches are only dealing with atomic changes which are assumed to occur in isolation and which can then be automatized in a pretty straightforward way. Complex modifications, which can be applied with arbitrary multiplicity and complexity, poses severe difficulties since they may present interdependencies which compromises the automation of the adaptation.

This work advocates the adoption of the transformational approach presented in [8] which encompasses the decomposition of difference models to distinguish among breaking resolvable and unresolvable metamodel changes. The main contribution of this paper is in providing a classification of the interdependencies which can occur in these two categories of modifications. The classification is used to define resolution criteria which provide the decomposition and the correct scheduling of modifications. Moreover, it has been shown how the dependencies are caused by features which are defined in the meta-metamodel (in this case KM3), which implies that the results are general and agnostic from the metamodel and its semantics.

Future works includes a more systematic validation of the approach, which necessarily encompasses larger population of models and metamodels. Finally, we plan to investigate how the works related to change impact analysis [21] can be adapted and used in MDE to support the co-evolution of metamodels and corresponding models.

References

1. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer **39**(2) (2006) 25–31
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
3. Favre, J.M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Procs. of the Int. Workshop ELISA at ICSM. (September 2003)
4. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: Proceedings of the 21st ECOOP. Volume 4069 of LNCS., Springer-Verlag (July 2007)
5. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Procs. of the 11th Int. Conf. MoDELS 2008, Toulouse (France). Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 326–340
6. Gruschko, B., Kolovos, D., Paige., R.: Towards Synchronizing Models with Evolving Metamodels. In: Procs of the Work. MODSE. (2007)
7. Herrmannsdoerfer, M., Benz, S., Jürgens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In: Procs. of the 11th Int. Conf. MoDELS 2008, Toulouse (France). Volume 5301 of Lecture Notes in Computer Science., Springer (2008) 645–659
8. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: 12th IEEE International EDOC Conference (EDOC 2008), Munich, Germany, IEEE Computer Society (2008) 222–231
9. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages & Computing **15**(3-4) (2004) 291–307
10. Del Fabro, M.D., Valduriez, P.: Semi-automatic Model Integration using Matching Transformations and Weaving Models. In: The 22th ACM SAC - MT Track, ACM (2007) 963–970
11. Bernstein, P.: Applying Model Management to Classical Meta Data Problems. In: Procs of the 1st Conf. on Innovative Data Systems Research (CIDR). (2003)
12. Galante, R., Edelweiss, N., dos Santos, C.: Change Management for a Temporal Versioned Object-Oriented Database. In: Procs. of the 21st ER. Volume 2503 of LNCS., Springer (2002) 1–12
13. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: FMOODS'06. Volume 4037 of LNCS., Springer-Verlag (2006) 171–185
14. 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)
15. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley (2003)
16. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6**(9) (October 2007) 165–185

17. Bézivin, J.: On the Unification Power of Models. Jour. on Software and Systems Modeling (SoSyM) **4**(2) (2005) 171–188
18. Di Ruscio, D.: Specification of Model Transformation and Weaving in Model Driven Engineering. PhD thesis, Università degli Studi dell'Aquila (February 2007) http://www.di.univaq.it/diruscio/phdThesis.php.
19. Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report n. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA) (April 2006)
20. Object Management Group (OMG): OCL 2.0 Specification (2006) OMG Document formal/2006-05-01.
21. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA (1996)