Abstract—Software development is witnessing the increasing need of version management techniques for supporting the evolution of model-based artefacts. In this respect, meta-models 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 meta-models have to be updated for preserving their well-formedness.

This work proposes an approach to raise the level of abstraction of meta-model adaptations by representing them as difference models conforming to a difference meta-model. This way, manipulations can be clearly distinguished from unchanged parts and semi-automated countermeasures can be taken in order to co-adapt the corresponding models.

I. INTRODUCTION

Increasingly, raising the level of abstraction beyond programming by describing problems directly using domain concepts is gaining acceptance in developing software systems. Model Driven Engineering [1] (MDE) leverages intellectual property and business logic from source code into high-level specifications enabling designers to focus on the crucial aspects of a system, which have traditionally been obscured by the usage of programming languages and underlying technologies. In general, an application domain is analysed and engineered by means of a meta-model, i.e. a coherent set of interrelated concepts. A model is said to conform to a meta-model, or in other words it is expressed by the concepts encoded in the meta-model, constraints are expressed at the meta-level, and model transformations are based on source to target meta-models.

The problem of evolution is an unavoidable phenomenon which concerns the whole life-cycle of a software system [2]. In fact, recording the various kinds of design-level structural evolution that a system undergoes throughout its life-cycle permits designers to grasp the underlying rationale and cannot be neglected in both software modelling and development [3]. 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. These different typologies influence the support mechanisms that will be required. Moreover, taxonomies of software evolution distinguish maintenance activities on the basis of their purpose (i.e. update, adaptive, performance, corrective or reductive) or by technical aspects (i.e., the when, where, what and how of software changes) [4], [5]. Therefore, evolution management is a complex task which requires specialized discipline and tool support.

Similarly to other software artefacts, meta-models can evolve over time too. Accordingly, models need to co-evolve in order to remain compliant with the meta-model, otherwise these artefacts may become invalid. When this operation is manually operated is error-prone and can give place to inconsistencies between the meta-model and related artefacts possibly leading to irremediable information erosion [6]. The proposed approach is based on a model difference representation [7] which is used to record in a difference model the meta-model changes. Thus, the co-adaptation is given as a higher-order model transformation which takes the difference model recording the meta-model evolution and produces a model transformation able to co-evolve the involved models.

The structure of the paper is as follows. In Sect. II the different kind of modification a meta-model can be subject to are illustrated and categorized in accordance with the available literature. Next section presents the different kinds of co-adaptation steps a meta-model evolution induces. Then the proposed approach is presented. Sect. V describes how to have a model-based representation of the meta-model evolution, whereas Sect. VI describes the automated co-adaptation. Finally, related works and some conclusions are discussed.

II. META-MODEL EVOLUTION

Modelling is at the heart of the Model Driven Engineering (MDE) idea [1]. It raises the level of abstraction of software development from code to models, with a great emphasis on focusing the developer concerns on the problem domain rather than on the underlying technologies. Meta-models can be considered one of the constituting concepts of MDE: 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. Meta-models are expected to evolve during their life-cycle, like any other software artefacts. As a consequence, changes may invalidate models which
conform to the old version of the meta-model and do not conform to the new version any more. The problem is due to the incompatibility between the meta-model revisions; a possible solution is the adoption of mechanisms of model co-evolution. More precisely, models need to be migrated in new instances according to the changes of the corresponding meta-model.

Unfortunately, model co-evolution is not always simple and presents intrinsic difficulties which are related to the kind of evolution the meta-model has been subject to. Going into more details, meta-models may evolve in different ways: some changes may be additive and independent to the other elements, thus requiring no or little instance revision. However, in other cases meta-model manipulations introduce incompatibilities and inconsistencies which can not be easily (and automatically) resolved. In particular, critical scenarios are characterized by information loss which can be recovered only through human intervention.

In Fig. 1 it is depicted an example of the evolution of a Petri Net meta-model. The initial Petri Net ($M_M^0$) consists at least of one Place and one Transition; moreover, places can have source and/or destination transitions, while transitions must link source and destination places (src and dst association roles, respectively). In a new meta-model $M_M^1$, arcs between places and transitions are made explicit by extracting $PTArc$ and $TPArc$ meta-classes. This refinement allows to add further properties to relationships between places and transitions. As $PTArc$ and $TPArc$ both represent arcs, they can be generalized by a superclass, which has been introduced in $M_M^2$ and named $Arc$. Finally, Petri Net formalism can be extended by annotating arcs with weights. Therefore, a new integer meta-property has been added to $Arc$ in $M_M^3$ revision of the meta-model, and it has named as $weight$.

The revisions illustrated so far can invalidate existing instances; therefore, each version needs to be analysed to comprehend the kind of updates it has been subject to and, eventually, the necessary adaptations of corresponding models. In the next Section, meta-model manipulations are classified by their corrupting or non-corrupting effects on existing instances.

### III. Model Co-evolution

Changes occurring on a meta-model may have different effects on the corresponding models. In this respect, meta-model changes are classified as follows [8]:

- **Not breaking changes**: changes occurring in the meta-model don’t break the models conformance to the meta-model;
- **Breaking and resolvable changes**: changes occurring in the meta-model do break the models, which can be automatically resolved;
- **Breaking and unresolvable changes**: changes do break the models and can not be automatically resolved and user intervention is required.

In particular, **not breaking changes** consist of additions of new elements in a meta-model $M_M^1$ leading to $M_M^{1'}$ without compromising models which conform to $M_M^1$ that in turn conform to $M_M^{1'}$. For instance, in the meta-model $M_M^2$ depicted in Fig. 1 the new class $Arc$ has been added as a generalization of the $PTArc$ and $TPArc$ meta-classes. After such a modification, models conforming to $M_M^1$ still conform to $M_M^2$ and co-evolution is not necessary. However, this is not always the case since in general changes break models even though sometimes automatic resolution can be performed because of breaking and resolvable changes. For instance, the Petri net meta-model $M_M^1$ in Fig. 1 is enriched with the new $PTArc$ and $TPArc$ meta-classes. Such a modification breaks the models that conform to $M_M^0$ since according to the new meta-model $M_M^1$, Place and Transition instances can not be directly related since $PTArc$ and $TPArc$ elements are required. However, such a breaking change can be automatically solved by adding for each couple of Place and Transition
elements two additional PTarC and TPArch instances among 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 MM3 in Fig. 1 which were not specified in MM2. The models conforming to MM2 can not co-evolve since only the designer 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.

Co-adaptation of models is strictly related to the notion of information preservation [6] from which the possible meta-model modifications can be distinguished into additive, subtractive, and updatable. In particular, additive changes consist of addition of new meta-model elements as described in the following:

- **meta-classes**: introducing a new meta-class in a meta-model is a common step in meta-model evolution. The result is in a super meta-model (a meta-model extension). To add a class involves an information loss only if that class is mandatory, in this case the instance model has to foresee it;
- **meta-properties**: a new meta-property may be or not obligatory. In the former case the change requires the introduction of the property in the involved class. Furthermore, if an addition of an abstract class without subclasses occurs, information is preserved;
- **meta-property generalisations**: a meta-property is generalised when multiplicity or type are relaxed. This involve new instances, but not new construction. It is information preserving;
- **Pulls of meta-properties**: a meta-property is pulled in the superclass and the old one is deleted in the subclass;
- **Extract superclass**: a superclass is extracted in a hierarchy and a set of property is pulled on. Only if the superclass is abstract the information is preserved, otherwise the effect is referable to the pull meta-property case.

Subtractive changes consists of the deletion of some of the existing elements as follows:

- **Eliminate meta-class**: a meta-class is deleted, the result is in a sub meta-model. This change is information preserving;
- **Eliminate meta-property**: a property is eliminated from a meta-class, it has the same effect of eliminate meta-class case. Information is preserved;
- **Push meta-property**: to push a property in subclasses means that it is deleted from the superclass and then cloned in all the subclasses. Information is preserved;
- **Flatten hierarchy**: to flat a hierarchy means eliminating a superclass and introducing all its property into the subclasses. Information is preserved;
- **Restrict meta-property**: a meta-property is restricted when multiplicity or type is reduced. It is a complex case into which 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 for the involved element usually manually provided. Restricting the type of a property requires type conversion for each value.

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

- **Rename meta-element**: renaming is a simple case into which the change need to be propagated but it is information preserving;
- **Move meta-property**: moving a property from a meta-class to an other ones can be easy propagated without loss of information;
- **Extract:inline meta-class**: to extract a meta-class means create a new class and move the relevant fields and methods from the old class into the new one. Instead, to inline meta-class means move all its features into another class and delete it. They are typical refactoring changes and are information preserving.

The classification illustrated so far is summarized in Table I. It becomes evident the fundamental role of model differences. Meta-model evolutions can be precisely categorized by understanding the kind of modification a meta-model undergone. Moreover, starting from the classification it is possible to adopt adequate countermeasures to co-evolve existing instances. In the sequel, the overall view of our proposal for the management of meta-model evolution and model co-evolution is illustrated.

<table>
<thead>
<tr>
<th>Change type</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Not breaking changes</strong></td>
<td>Generalize meta-property</td>
</tr>
<tr>
<td></td>
<td>Add (non-obligatory) meta-class</td>
</tr>
<tr>
<td></td>
<td>Add (non-obligatory) meta-property</td>
</tr>
<tr>
<td><strong>Breaking and resolvable changes</strong></td>
<td>Extract (abstract) superclass</td>
</tr>
<tr>
<td></td>
<td>Eliminate meta-class</td>
</tr>
<tr>
<td></td>
<td>Eliminate meta-property</td>
</tr>
<tr>
<td></td>
<td>Push meta-property</td>
</tr>
<tr>
<td></td>
<td>Flatten hierarchy</td>
</tr>
<tr>
<td></td>
<td>Rename meta-element</td>
</tr>
<tr>
<td></td>
<td>Move meta-property</td>
</tr>
<tr>
<td></td>
<td>Extract:inline meta-class</td>
</tr>
<tr>
<td><strong>Breaking and unresolvable changes</strong></td>
<td>Add obligatory meta-class</td>
</tr>
<tr>
<td></td>
<td>Add obligatory meta-property</td>
</tr>
<tr>
<td></td>
<td>Extract superclass (non-abstract)</td>
</tr>
<tr>
<td></td>
<td>Pull meta-property</td>
</tr>
<tr>
<td></td>
<td>Restrict meta-property</td>
</tr>
<tr>
<td></td>
<td>Extract (non-abstract) superclass</td>
</tr>
</tbody>
</table>

**TABLE I**

CHANGES CLASSIFICATION

**IV. PROPOSED APPROACH**

In MDE, it is of critical relevance that designers are able to comprehend the various kinds of designlevel structural evolution that a system undergoes throughout its entire life-cycle. The detection of differences between models is essential to model development and management practices. They can
be exploited for deriving artefacts or drawing conclusions on the rationale behind the modifications. Meta-models evolve like other artefacts; in a context adhering to the “everything is a model” principle [9], meta-models can be considered as models and their evolution can be managed in an analogous way. Therefore, the problem of meta-model evolution can be tackled in a high level way by using a difference meta-model together with the related change representation mechanism. A document representing the differences occurred between two meta-model versions include all the necessary information to detect the change typology and consequently how it affects model instances. Starting from the classification of meta-model evolutions discussed in Section III, corresponding distinctions can be made between difference elements, which can be categorized depending on the kind of meta-model adaptation they represent. Hence, delta elements induce transformation rules able to migrate models from a version to another, as depicted in Fig. 2. Changes occurred between the initial meta-model $MM_1$ and the new meta-model $MM_2$ are represented in a difference document $\Delta_{MM}$, which conforms to the difference meta-model $MM_{\Delta}$. $MM_{\Delta}$ is obtained by an automated extension of the meta meta-model taken into account. All the meta-models conform to such meta meta-model.

![Fig. 2. Models co-evolution using model differences](image)

In the following Section, the adopted approach to difference representation is presented.

V. REPRESENTATION OF META-MODEL DIFFERENCES

As illustrated so far, calculation and representation of model differences are essential to model development and management practices. In [7] we proposed a number of properties on representation techniques to leverage model differences to first class entities pursuing the “everything is a model” [9] principle. In particular, a difference representation should be:

- **model-based**, the outcome of a difference calculation must be represented as a model to enable a wide range of possibilities, such as subsequent analyses, conflict detection or manipulations;
- **compact**, the difference model must be compact and contain only the necessary information to represent the modifications, without duplicating parts as those model elements which are not involved in the modification;
- **self-contained**, a difference model must not rely on external sources of information, as for instance references to base model elements or base meta-models;
- **transformative**, each difference model must induce a transformation, such that whenever applied to the initial model yields the final one; moreover, the transformation must be fuzzy, i.e. applicable also to any other model which is possibly left unchanged in case the elements specified in the difference model are not contained in it;
- **compositional**, the result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations;
- **meta-model independent**, the representation techniques must be agnostic of the base meta-model, i.e., the meta-model the base models conform to. In other words, it must be not limited to specific meta-models, as for instance happens for calculation methods given for the UML meta-model.

![Fig. 3. KM3 meta-model](image)

In the following, we outline the metamodel-independent approach to difference representation already presented in [7], which satisfies the requirements summarized above. The contribution of this work relies on such model-based representation: differences between meta-model versions are stored as difference documents denoting the manipulations the meta-model undergone during its life-cycle. Moreover, deltas are exploited to co-evolve existing models conforming to the old version of the meta-model. Despite the work in [7] has been introduced to deal with model revisions, it is easily adaptable to meta-model evolutions too. In fact, a meta-model is a model itself, which conforms to a meta-model referred to as the meta-meta-model. The discussion will be supported by the running example already introduced in the previous Sections. For presentation purposes, the KM3 meta-model in Fig. 3 is considered throughout the paper even though the approach is general and is applicable to any meta-model like the OMG/MOF [10] or EMF/ECore [11].

The overall structure of the change representation mechanism is depicted in Fig. 4: given two meta-models which
conform to an arbitrary meta meta-model (KM3 in our case), their difference conforms to another meta-model (MMD) derived from the former by means of an automated transformation (MM2MMD). The initial meta-model, extended as prescribed by such a transformation, consists of new constructs able to represent the possible modifications that can occur on meta-models and which can be grouped as follows:

- **additions**: new elements are added in the initial meta-model. With respect to the classification given in Section III, 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 meta-model being considered can consist of updates of already existing elements. For instance, Rename metaclass and Restrict metaproperty require this type of modification. Also the addition and deletion of metaproperty (i.e. Add metaproperty and Eliminate metaproperty, respectively) are modelled through this construct. In fact, when a metaclass is included in a container the manipulation is represented as a change of the container itself. This way the information about the container can be preserved to satisfy the self-containedness of the representation [7].

The meta-models in Fig. 1 have been deliberately kept simple without loss of generality because of space limitation. The KM3 language (see Fig. 3) is based on analogous core concepts used in OMG/MOF and EMF/Ecore and is focused on meta-modelling only (that is, Java code generation facilities are not supported, for instance). A number of experimental KM3 meta-models have been specified both from academia and industry and are currently collected into a library that can be found at [12]. Furthermore, the available tool support is able to generate Ecore and MOF meta-models corresponding to the given KM3 specifications. Finally, the models in Fig. 1 are suitable to present all the insights of the co-adaptation mechanisms as already demonstrated in [6].

In order to represent the differences between the Petri Net meta-model revisions, the extended KM3 meta-model depicted in Fig. 5 is generated by applying the MM2MMD transformation in Fig. 4 previously mentioned. For each meta-class MC of the KM3 meta-model, the additional meta-classes AddedMC, DeletedMC, and ChangedMC are generated. For instance, the meta-class Class in Fig. 3 induces the generation of the meta-classes AddedClass, DeletedClass, and ChangedClass as depicted in Fig. 5. In the same way, Reference metaclass induces AddedReference, DeletedReference, and ChangedReference.

The generated difference meta-model is able to represent all the differences amongst meta-models which conform to KM3. For instance, the model in Fig. 6 conforms to the generated meta-model in Fig. 5 and represents the differences between the Petri Net meta-models depicted 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 $MM_1$ revision of the Petri Net meta-model is represented by means of an AddedClass instance, as illustrated by $MM_1 - MM_0$ delta in Fig. 6. 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. Finally, two new
references has been added through the corresponding AddedReference instances to realize the reverse links from PTArc to Place and Transition, respectively; 2) the addition of the new abstract class Arc in $MM_2$ is represented through a change of the AddedClass meta-class in the $MM_2 - MM_1$ delta of Fig. 6. In the meantime, PTArc and TPArc classes are made specializations of Arc; 3) in $MM_3$ version of the Petri Net meta-model the new weight property is added to Arc by means of an instance of ChangedClass, which updates Arc with the new attribute.

The representation mechanism used so far allows to identify changes which occurred in a meta-model revision. Its self-containedness permits to re-apply the same manipulation also to meta-model revisions differing from the original one. Moreover, the use of the abstract syntax eases the customization of modification layouts in order to highlight updates in a user-friendly way. Finally, difference models can be exploited to manage model co-evolution induced by meta-model manipulations as illustrated in the next Section.

VI. MANAGING BREAKING CHANGES

The migration of models can be derived from difference models representing the updates which occurred. In fact, each manipulation of a meta-model conforming to KM3 can be distinguished with respect to the classification presented in Section III. For example, each AddedClass without any relationship with the existing meta-model is considered as a non-breaking change. On the contrary, each ChangedAttribute inducing the movement or the deletion of a meta-property for instance, is a breaking and resolvable adaptation. Finally, ChangedReference and ChangedAttribute modifications which enforce StructuralFeature constraints are breaking and unresolvable. Examples of this scenario happen when the lower property value is increased, the upper value is decreased, or the isUnique flag is enabled.

The co-adaptation of models induced by the meta-model evolution can be reduced to these three cases, which are considered in three subsequent steps, as shown in Fig. 7. $MM_1$ is the initial meta-model which conforms to KM3; after manual changes $MM_2$ is obtained. The difference document $\Delta_{MM}$ is decomposed in a $\Delta_{MM}[\text{resolvable}]$ model, containing the resolvable changes, and a $\Delta_{MM}[\text{unresolvable}]$, with the unresolvable ones. Hence, the co-evolution of existing models is managed by splitting updates as follows:

1) non-breaking changes do not need to be propagated and are ignored;
2) breaking and resolvable modifications are taken into account by automated transformations able to interpret $\Delta_{MM}[\text{resolvable}]$ and co-adapt existing instances, as illustrated in Section VI-A;
3) unresolvable changes present some difficulties. In fact migration rules cannot be directly derived from delta models ($\Delta_{MM}[\text{unresolvable}]$ in this case) and human intervention is necessary to provide further information. Nonetheless, also in this situation some degree of automation can be achieved through transformation induction techniques, as illustrated in Section VI-B.
A. Model co-evolution induced by breaking and resolvable changes

<table>
<thead>
<tr>
<th>Manual change</th>
<th>Modification in the model difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract (abstract) superclass</td>
<td>AddedClass, ChangedClass</td>
</tr>
<tr>
<td>Eliminate meta-class</td>
<td>DeletedClass</td>
</tr>
<tr>
<td>Eliminate meta-property</td>
<td>ChangedClass, ChangedProperty</td>
</tr>
<tr>
<td>Push meta-property</td>
<td>DeletedClass, ChangedClass</td>
</tr>
<tr>
<td>Flatten hierarchy</td>
<td>ChangedClass, ChangedProperty</td>
</tr>
<tr>
<td>Rename meta-element</td>
<td>ChangedClass</td>
</tr>
<tr>
<td>Move meta-property</td>
<td>ChangedClass, ChangedProperty</td>
</tr>
<tr>
<td>Extract meta-class</td>
<td>ChangedClass, AddedClass</td>
</tr>
<tr>
<td>Inline meta-class</td>
<td>ChangedClass, DeletedClass</td>
</tr>
</tbody>
</table>

TABLE II

CHANGES REPRESENTATION

Breaking and resolvable changes are easily manageable by definition. In general, they involve refactorings of meta-models, like meta-class renaming or meta-property movement, which require simple co-adaptations.

Starting from the difference document, transformation rules can be given to solve these kinds of co-evolution issues. In particular, each manual change classified as breaking and resolvable can be identified through a representation in the difference model, as shown in the Table II. For example, the extraction of a superclass is depicted by means of an AddedClass instance together with ChangedClass instances related to the generalized meta-classes. In a similar way, a meta-class renaming is represented through a ChangedClass element.

From difference model entities representing meta-element updates, the migration rules for the corresponding instances can be derived. For instance, in Fig. 8 QVT rules [13] are depicted to manage resolvable meta-model adaptations involving References. Going into more details, when a reference name is changed in the meta-model, it has to be updated in all its existing instances. In the same way, a reference type (i.e. the pointed meta-class) manipulation has to be kept up to date in existing models.

By recalling the example in Fig. 1, the changes occurred between $MM_0$ and $MM_1$ meta-models can be automatically resolved by the transformation rules illustrated in Fig. 8 applied to the broken models. Arcs between Places and Transitions are made explicit by extracting PTArc and TPArc meta-classes. The Reference named dst and owned by Transition is changed in out and directed to PTArc. Vice versa, the Reference named src owned by Place is changed in in and directed to PTArc. The same rule is applied with the same role also for the class TPArc. The patterns contained in the QVT rules shown in Fig. 8 match delta models and Petri Net instances as summarized in Fig. 9.

B. Semi-automatic model co-evolution induced by breaking and unresolvable changes

The difference model $\Delta_{MM[unresolvable]}$, representing the only changes non-autonomously resolvable, is used to co-evolve the $M_2$ model, in which resolvable changes have been propagated by the mechanism presented in the previous Section. As already mentioned, unresolvable changes can not be directly propagated in the instance models because of lack of information. For instance, if a new mandatory class is introduced in the meta-model, at least default values should be provided in order to add instances of such class in the models. In particular, changes of this type may require:

- Selection criteria: property or multiplicity restriction can require the deletion of some instances. Therefore, selection criteria are exploited to specify, or at least to suggest, which choice to make;
- Default values: some changes (i.e. addition of a new mandatory element, restriction of a property or a type) cause instances to be added in the corresponding model. Consequently, instance values are needed and defining default values can be a possible solution;
- Conversion functions: Restricting the type of a property requires type conversion for each value.

Breaking and unresolvable category requires human attention. Developers will have to provide additional information
needed to migrate the affected instances [8]. In this paper, human intervention is used in order to collect information as knowledge base exploitable to obtain semi-automatic co-evolution of models also in case of critical changes. Going into more details, we propose to support co-evolution of models needing designer decisions by using inductive logic programming (ILP) [14]. ILP can be defined as an intersection of inductive learning and logic programming. The examples background knowledge and final descriptions are all described as logic programs, thus using induction instead of deduction as the basic mode of inference.

Our proposal takes inspiration from Model Transformation by-example (MTBE) [15] approach, which consists of deriving model transformation rules from an initial prototypical set of interrelated source and target models. In general, the initial (or base) knowledge describes critical cases of the model transformation problem in a purely declarative way. In this respect, we consider a variety of unresolvable changes and change instances, corresponding to different solutions, as facts which constitute our knowledge base. An inductive approach, which proceeds backwards and abducts hypothesis from some facts, is able to generate co-evolutive transformation rules. Our engine learns from changes already resolved by hand-tuning choices and improves the results with subsequent refinements.

The ILP system used to this purpose is A Learning Engine for Proposing Hypotheses (Aleph) [16]. It requires three files to construct theories, i.e. a background knowledge file, a positive rules supporting the knowledge base are shown. The background knowledge and final descriptions are all described as Prolog clauses that encode information relevant to the domain.

For example, in the $MM_3$ of Fig. 1 a new attribute weight has been added to the Arc class. Hence, a mandatory attribute has to be added in all Arc subclasses, and consequently in all the corresponding instances which need to be migrated from $MM_2$ to $MM_3$ meta-model. This issue could be resolved by using default values, as explained above. In the Listings 1, 2 fragments of example models are encoded. While in the source model changes have not been solved, in the target model, the new attribute weight has been added and valued 1 as default.

```
1 metaclass(class(arch(a1))).
2 metaclass(class(net(n1))).
3 metaprop(prop(weight(w1))).
4 metaprop(prop(name(p1))).
5 metaedge(edge(arch(a1),weight(w1))).
6 class(arch(a1)).
7 class(net(n1)).
8 prop(weight(w1)).
9 prop(name(p1)).
10 edge(arch(a1),weight(w1)).
```

Listing 1. Source model encoding

```
prop_value(weight(w1),1).
default(1).
```

Listing 2. Target model encoding

In Listing 3 auxiliary clauses representing the differences between meta-models are encoded, while in Listing 4 basic rules supporting the knowledge base are shown.

```
1 class(X) :- X.
2 prop(X) :- X.
3 prop_value(X,Y) :- prop(X), default(Y).
4 edge(X,Y) :- X, Y.
```

Listing 3. Metamodel differences encoding

In the following Listing some positive and negative clauses are encoded.

```
1 prop(weight(w1)).
2 edge(arch(a1),weight(w1)).
3 prop_value(weight(w1),1).
```

Listing 4. Basic rules supporting the knowledge base

```
1 prop(arch(a1)).
2 edge(arch(a1),arch(a1)).
3 prop_value(name(p1),1).
4 prop_value(arch(a1),1).
```

Listing 5. Positive clauses

```
prop(A) :- add_attribute(A), metaprop(A),
edge(A,B) :- class(A), prop(B), add_attribute(B),
update_element(A), metaprop(A),
prop_value(A,B) :- prop(A), default(B), add_attribute(A).
```

Listing 7. Induced transformation rules

Unresolvable changes can be supported in semi-automatic way, this means that designers can be addressed to their choices through different class of rules, each suitable for a particular kind of modification needing human intervention. The approach operates in a sequential way, that is at each step new instances of the example model are generated and refined by means of hand-tuning. In fact, a larger background knowledge increases the possibility to automate the process and to induce correct mappings.

VII. RELATED WORKS

Meta-model evolution and model co-evolution issues have been explored by several works like [8], [6]. In general, model co-evolution is tackled by categorizing changes as a) without effects on existing model instances, b) with simple side effects on models, c) with side effects requiring further information to be managed [17]. In this respect, the change classification made in Section III takes inspiration from the existing experience on meta-model evolution management. In [6] the author distinguishes between adaptations
that ensure instance preservation and manipulations which induce co-evolutions. Moreover, he defines QVT transformations to co-adapt instances when resolvable changes occur. With respect to [6], this work exploits the benefits of a suitable model difference representation which gives the possibility to reapply the represented modifications to meta-models different to those considered for the difference calculation (see the self-contained property in Section V). Moreover, the representation approach allows to decompose the modifications in separate models according to the categorization given in Section VI.

The classification by non-breaking and breaking changes has been introduced in [8], where the authors use the Epsilon Transformation Language (ETL) to migrate models from a meta-model revision to another one. The mechanism for meta-model evolution representation is not explicitly specified, even though the problem of change detection is discussed. In fact, the authors suggest the use of changes trace as opposed to direct comparison, in order to be able to detect more complex manipulations like element movements. Furthermore, the solution only deals with resolvable changes, which are managed through ETL transformation facilities and user input gathering. Therefore, with respect to our approach changes are not precisely encoded and consequently a classification of meta-models updates is not provided, while resolvable changes are deferred to future research. Moreover, the use of inductive logic techniques proposed in this work makes the users not to write transformation rules; whereas, they have only to define a set of model mappings which are exploited to induce new transformation rules and eventually to refine existing ones.

The problem of model differences can be considered as made up of three relevant concerns, i.e. their calculation, representation, and visualization. In current approaches these three aspects tend to partly overlap; for example, the calculation is designed depending on the kind of modifications that will be represented. Moreover, in general the visualization is a simple graphical arrangement of the difference information obtained through the calculation. We believe that a model-based representation mechanism is necessary to enable evolution analysis and management in MDE [7]. In this respect, this work constitutes a demonstration of the need and usefulness of such a mechanism.

Transformation by example is a current research field emerged by a simple observation: domain experts know the domain of the application and not how to write transformations. ILP techniques have been proposed to derive both graph transformations [15] and ATL rules [18]. This paper exploits ILP mechanisms to induce the automated co-evolution of resolvable changes. In our case, mappings can be derived through the countermeasures adopted for resolvable updates and the first developers’ resolutions for unresolvable situations. Moreover, when some hand-tuning occurs the transformation is refined coherently with the new knowledge.

VIII. Conclusion and Future Works

In software development versioning techniques are increasingly needed for supporting the evolution of model-based artefacts. The detection and representation of differences between models is essential to model development and management practices. For example, delta documents can be exploited to highlight the manipulations a system undergone or to generate refactoring actions in order to let a given system migrate.

This work presented a mechanism for the management of meta-model evolution in a MDE environment: changes are represented in a model-based fashion to enable model-driven techniques. Difference models are exploited to precisely classify meta-model manipulations and consequently to adopt corresponding countermeasures to co-adapt existing instances. This way, a portion of model migration can be completely automated. Some scenarios still require human intervention because of lack of information. However, it is sufficient to provide a minimal set of decisions to make the mechanism able to coherently induce the remaining co-adaptations. Moreover, the developer can hand-tune obtained mappings to refine the generated resolutions.

We believe that improving the degree of co-adaptation automation becomes of fundamental importance when dealing with large projects. In fact, migration activities can be time-consuming and error-prone, and in the meantime driven by precise adaptation patterns. Transformation by-example has been demonstrated as a profitable approach for mapping specification [15], [18]. Nonetheless, this proposal needs to be empirically validated against complex systems, which requires the implementation of the whole mechanism in a modelling platform like AMMA [19], EMF [11] or GME [20]. Further work will also encompass the investigation of transformation co-evolution. In fact, model transformations are specified by means of source and target meta-models. Therefore, when a meta-model is changed also transformation definitions need to be migrated.

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 meta-models and corresponding models. In fact, change impact analysis provides techniques and tools to assess the impact of software changes and gather information to accordingly modify the design of the considered software system. Even though these techniques work at a different level of abstraction, they might be generalized and adopted to deal with the model co-evolution problem.

REFERENCES


