Using logic programming for adapting models to metamodel evolution

Fouzia Anguel*

Chadli Bendjedid University,
BP 73, El Tarf, 36 100, Algeria
and
LISCO Laboratory,
Badi Mokhtar University,
B.P. 12, Annaba, 23000, Algeria
Email: fanguel@yahoo.fr
*Corresponding author

Abdelkrim Amirat

LiM Laboratory,
Mohamed Cherif Messaadia University,
Souk Ahras, 41000, Algeria
Email: abdelkrim.amirat@yahoo.com

Nora Bounour

LISCO Laboratory,
Badi Mokhtar University,
B.P. 12, Annaba, 23000, Algeria
Email: nora_bounour@yahoo.fr

Abstract: Evolution is inherent in software systems because of the rapid improvement of technologies. As metamodels are the cornerstone of model driven engineering and they evolve iteratively, their evolution affects the rest of artefacts involved in a development process, e.g., models or transformation rules. Therefore, co-evolution tools for models and other artefacts are indispensable. We present in this paper an intelligent approach to adapt models by means of state-based and operator-based techniques. The defined co-evolution process consists of four phases. Initially, changes between two metamodel versions are detected. After that, evolution scenario is reconstructed using logic programming method. Evolution scenario is first calculated from difference model and represented by a set of primitive evolution operations, after that it is transformed using an inference engine to include eventual composite evolution operations; from this scenario adaptation solutions are generated according to the evolution operator impact in model level. Finally, migration scenario is applied on an input model conforming to the old metamodel version to remain compliant with its metamodel.

Keywords: metamodel evolution; model migration; logic programming; primitive operator; composite operator; evolution operator; evolution scenario; coupled operator.
1 Introduction

Metamodels and domain-specific languages are key artefacts in model driven engineering (MDE) (Bézivin, 2005), as they are used to define syntax and semantics of domain models. Metamodels are in continuous evolution. Thereby, different versions of the same metamodel are created and must be managed (Favre, 2003). The evolution of metamodels is relatively a broad problem because many artefacts are related to it namely models/transformation rules and might be impacted by metamodel changes. Therefore metamodel evolution may require the migration of their related artefacts. In this work we tackle the problem of models migration. This problem is referred to as metamodel evolution and model co-evolution (Wachsmuth, 2007), model adaptation (Garcés et al., 2009) or model migration (Rose et al., 2010). We will use these terms indifferently. Several approaches enabling model migration have been proposed. Each approach aims to reduce the effort required to perform metamodel and model co-evolution. Rose et al. (2009) propose a classification of model migration approaches. This classification highlights three ways to identify needed model updates:
Using logic programming for adapting models to metamodel evolution

Manual approaches provide transformation languages to manually specify model migration. State-based approaches use metamodel matching to deduce differences which are used to semi automatically derive a transformation that expresses models adaptations. However, operator-based approaches consist on recording evolution trace when changing the metamodel and thus intended model migration is captured from this trace. Most of existing approaches integrate model designer in the evolution process. However, model designer is a user (i.e., metamodel user) who cannot always comprehend the intention behind the evolution process; he must be only responsible of model migration. This is the first challenge. Our overall goal is to propose assistance to the user in order to manage metamodel evolution and particularly manage impacts on models. In the other hand existing approaches have advantages but present also some drawbacks. Herrmannsdoerfer and Wachsmuth (2014) affirmed that no matured state-based approach is developed until know. Furthermore, the major drawback of operator-based approaches is the coupling between the migration tool and the recorder tracking metamodel changes. Therefore, we have defined a second challenge which is automating as far as possible migration process.

In this paper, we propose an approach for designing and implementing tool components which can adapt models to evolved metamodel version. For this purpose, we opted for a hybrid approach using at the same time matching techniques to identify primitive differences between two metamodel versions. These differences are further processed by an inference engine which identifies sets of atomic operations that implement a composite evolution operation, and determines these operations as well as their parameters. These results are used to adapt models.

In our approach, the ‘reconstruction of evolution operations’ tool component is implemented using Prolog (Savoy, 2006). We use a library of coupled evolution operators encoded in logic programming formalism (LP). Models and differences are also encoded in LP. Finding groups of primitive operations is basically a pattern-matching problem which is solved by the matching engine of Prolog.

The rest of the paper is structured as follows. Section 2 describes metamodel and model co-evolution problem. Section 3 discusses existing works. Section 4 explains our proposed approach for solving the model co-evolution problem. Finally, we conclude the paper in Section 5.

2 Background

2.1 Meta-modelling formalism

In MDE, modelling languages are used to describe a set of models. The abstract syntax of a modelling language is described by a metamodel (Bézivin, 2005). A metamodel is a model that defines concepts for expressing models. Models enable to represent a system at a higher level of abstraction. Analogously, metamodels are defined by using concepts described as meta-metamodels. This hierarchy is specified by the Object Management Group (OMG) (2015a) in four-layer modelling architecture. Thus, in MDE everything is
considered as model (Bézivin, 2005). Metamodels can be expressed in various meta-modelling formalisms. Well-known examples are the Meta Object Facility (MOF) (OMG, 2015b), the meta-modelling standard proposed by OMG and Ecore which is the meta-modelling formalism underlying the Eclipse Modelling Framework (EMF) (2015a). MOF was originally derived from a subset of UML (OMG, 2015b) and shares many common modelling elements with UML. The main elements are called classes, which consist of a number of features. Classes can have super-types to inherit features and might be abstract. A feature has a multiplicity (lower and upper bound) and is either a property or an association. A property is a feature with a primitive type whereas an association is a feature with a class type. An association may be composite and two associations can be opposite. In the MOF metamodel, the concept of inheritance is called generalisation. Figure 1 shows an excerpt of the MOF model. The MOF model is described in itself. At the implementation stage, we use meta-metamodel Ecore from the EMF (2015a). However, our approach is not restricted to Ecore, as it can be transferred to all object-oriented meta-modelling formalisms.

**Figure 1** Essential MOF classes

![MOF model diagram](source:OMG (2015a))

In this paper we provide a formal representation of metamodels and models in terms of predicates and sets. Thus, we translate the representation of MOF metamodels in LP clauses in order to be used later by an intelligent reasoning mechanism.
2.2 Logic programming

Knowledge-based systems are used as a way to store and manipulate knowledge to interpret information in a useful way. They are often used in artificial intelligence applications and research. Rules are the basic form of knowledge representation in many areas of artificial intelligence, including LP. This programming paradigm is based on logic (more accurately, the predicate calculus). A LP program is a set of sentences in logical form, expressing facts and rules about some domains (Baral and Gelfond, 1994). Major LP language use rules written in the form of clauses (Savoy, 2006) as presented in equation (1) and are read declaratively as logical implications: ‘H if B1 … and Bn’. Facts specify statements or relationships which are held to be true. They are rules that have no body, and are written in the simplified form: ‘H’. The idea of using logic to write computer programs is to specify what the programs should do and not how they should do it. In LP, the specification for a program is written, and the computer executes it. A LP program executes a goal whose answer is a conjunction of conditions. Goal resolution is automated and generally performed by LP solver (Baral and Gelfond, 1994). For the definition of rules specifying the knowledge base used in our approach, we use the first and most important LP language Prolog (Savoy, 2006). Prolog is chosen because in one hand it is a language of knowledge representation and in the other hand Prolog interpreters are developed in several languages, which facilitate the use of the Prolog formalism. We notice that in this paper we use Prolog notation for clauses: predicate symbols, function symbols and constants start with a lower case letter, and variables start with an upper case letter or underscores (‘_’) if their values are indifferent. Numbers can be treated as constants. The predicates may have arguments.

\[ H : \neg B_1, \ldots, B_n. \]  

(1)

2.3 Metamodel and model co-evolution

In fact, metamodels undergo complex evolutions during their life cycles (Favre, 2003). As a consequence, when a metamodel is modified, models conforming to this metamodel need to be migrated in such a way they conform to the modified version as depicted in Figure 2. Therefore, models need to co-evolve in order to remain compliant with their metamodels. Without co-evolution, these artefacts become invalid or obsolete. To manage models co-evolution, it is important to comprehend metamodel evolution which is defined as a set of changes applied on the initial metamodel. A number of works proposed the classification of metamodel changes according to their corrupting effects. Generally, the more used changes classification is that proposed in Gruschko et al. (2007) where metamodel changes are grouped on three categories:

- not breaking changes: changes occurring in the metamodel do not break models conformance to the metamodel
- breaking and resolvable changes: changes occurring in the metamodel do break models conformance, which can be automatically resolved
- breaking and non-resolvable changes: changes do break the models and cannot be automatically resolved and user intervention is required.
However, a uniform formalisation of metamodel evolution is still lacking (Rose et al., 2009). The relation between metamodel and model changes should be formalised in order to allow reasoning about the correctness of model migration definitions. To deal with this problem, we propose a representation of metamodel changes in LP formalism.

Throughout this paper, we use a Petri net metamodel (Wachsmuth, 2007) as a running example. Figures 3(a) and 3(b) show the metamodel before and after its evolution as a UML class diagram MM1 and MM2 respectively. In Petri net metamodel (MM1), a net consists of any number of places and transitions. Each transition has at least one input and one output place. Since a Petri net without any places and transitions is of no avail,
the new version of the metamodel (MM2), shown in Figure 3(b) restricts net to comprise at least one place and one transition, by changing lower bounds in containment references of MM1. Furthermore, MM2 makes arcs between places and transitions explicit by introducing PTArc and TPArc classes. As a consequence, references between place and transition disappear since these relationships are explicitly represented by references with PTArc and TPArc. As well, because PTArc and TPArc both represent arcs, they are generalised by arc class. Besides, arcs might be annotated with weights. This extension is reproduced by introducing a new attribute weight in arc class. Finally, MM2 incorporates a new class token and its related containment relationship with place class to cover dynamic aspects of Petri nets by marking places with tokens.

The new Petri net metamodel version invalidates existing models. Thus, we focus in this work on modifications done over Petri net metamodel MM1 and their impact over conforming models.

3 Related works

In this section we will give an overview of current metamodel and model co-evolution approaches and already implemented systems. The necessity of support for efficient and automatic co-evolution of metamodel and models was identified as a major challenge in software evolution (Mens et al., 2005). Currently, there are several approaches that support this problem; Rose et al. (2009) propose a classification of model migration approaches. This classification highlights three ways to identify needed model updates:

1. manually
2. based on operators
3. state-based approaches.

Manual approaches, like Sprinkle and Karsai (2004), Narayanan et al. (2009) and Rose et al. (2010), provide transformation languages to manually specify the model migration. Sprinkle and Karsai (2004) present a visual model transformation language to specify the transformation only for the difference between two metamodel versions. This language is extended resulting model change language (MCL) which uses patterns for typical migration scenarios (Narayanan et al., 2009). Flock is an EMF-based textual model transformation language that also automatically unsets model elements that are no longer conforming to the adapted metamodel (Rose et al., 2010). In operators-based approaches (Wachsmuth, 2007; Herrmannsdoerfer, 2010) metamodels changes are defined in terms of co-evolutionary operators. Those operators define conjointly the evolution on the metamodel and encapsulate model migration. Wachsmuth (2007) shows an operation-based tool prototype called ECORALL that combines object-oriented refactoring and grammar adaptation to provide a library of reusable-coupled operations a basis for automatic metamodel evolution. Herrmannsdoerfer (2010) developed COPE which explicitly records the history of the metamodel as a sequence of changes and allows attaching information on how to migrate models. Attached information can be used to migrate automatically models to the new metamodel version. To increase expressiveness, COPE extends the approach proposed in Wachsmuth (2007) with a means to specify a custom model migration for a recorded metamodel adaptation. Finally,
state-based approaches use metamodel matching. They are only based on the state of the metamodels which are compared and differences between them are used to semi-automatically derive a transformation that expresses models updates (Cicchetti, 2008). Gruschko et al. (2007) classify primitive metamodel changes into not breaking, breaking resolvable and breaking non-resolvable and envision to automatically detecting of model migration for breaking resolvable changes. Moreover, Cicchetti et al. (2007) have proposed capturing differences between models through the use of a difference metamodel and present a tool prototype based on Eclipse EMF that supports model evolution for primitive changes and compound changes. In Garcès et al. (2009), authors developed an EMF-based matching language that supports customisation of the matching process and adding new matching patterns.

After this brief survey of existing approaches for metamodel and model co-evolution, we can discern that while manual approaches foster correctness of the model migration, they also require the most effort. Matching approaches try to completely automate the building process but they may not always lead to a correct model. As for operators-based approaches lead to a correct migration by means of recording evolution trace at the same time and reduce the effort by reusing coupled operators. However, they also require integration into the editor for the metamodel. Analysis of advantages and disadvantages of existing approaches helps us to motivate our approach targeting to adapt models to their evolved metamodels. In this work, we propose an alternative solution where we use a mixture of state-based and operator-based approaches to adapt models to metamodel evolution. Our approach goes beyond existing works in three directions.

Firstly, in existing approaches we notice a strong involvement of model designer in evolution process, although he is not always connoisseur of metamodel design. He must be able to evolve metamodel, or identify differences between two versions of the metamodel, which is not easily accomplishable. Our view is different; a metamodel can have different users. We estimate that model designer is generally different from metamodel designer. Our proposed approach is oriented to model designer (i.e., user) and aims to provide to users a tool support to accomplish successfully the migration of their models without participating to metamodel evolution process. The objective is to reduce as much as possible the different interventions of the model designer and to guide the co-evolution process. This is why; we consider that metamodel changes are calculated posteriorly as in state-based approaches.

Secondly, since operator-based approaches gave good results (Herrmannsdoerfer and Wachsmuth, 2014), and Specially COPE with its library covering a large set of operators (Herrmannsdoerfer et al., 2011). We envisage using operators in our approach, however we have find that COPE approach is oriented implementation where evolution operators are described by coded functions to be executed and they do not have a formal definition and are not documented. In contrast, our approach proposes a formalisation of a set of atomic and compound operators in order to use them more efficiently and to facilitate eventual library extension. The proposed library provides a high degree of expressiveness to this approach by adding logical structure.

Thirdly, correct evolution management needs considering both atomic and compound evolution operations. In our proposal the novelty is the use of an intelligent mechanism to reason about evolution operations in order to reconstruct compound operations and therefore to got an evolution scenario. Currently in the best of our knowledge there is no approach that uses an intelligent reasoning to resolve metamodel and model co-evolution problem.
4 Proposed approach

In this section we describe our proposal to perform the co-evolution of models with their evolved metamodels. The overall evolution and co-evolution process is presented in Figure 4.

Our approach is hybrid and it imports techniques from state-based and operator-based approaches and uses also a reasoning mechanism from artificial intelligence. It contains four phases:

1. detection of changes
2. reconstruction and validation of evolution scenario
3. determination of migration scenario
4. running of model migration.

In the first step; differences between two metamodel versions need to be determined. In the second step we reconstruct atomic evolution operations from primitive changes then we use an inference engine to generate evolution scenarios by assembling atomic operations in possible compound ones; in the third step we explore a library of operators to obtain different migration procedures, which will be assembled to constitute migration scenario. In the last step the migration scenario will be applied over a specific model conforming to the old version in order to obtain a new model conforming to the newer metamodel version. During this step users decide how the model will change based on the possible alternative solutions. With this co-evolution process, we define a preliminary phase to encode metamodels, changes and a library of operators using LP formalism. Resulted knowledge base is the core of the co-evolution process.

Figure 4  Overview of co-evolution process

4.1 Preliminary phase

This phase consists on preparing environment to execute proposed co-evolution process. It is performed through three steps. Firstly, we propose a formalisation of MOF-based metamodels and models in LP. Secondly, we propose an encoding method to represent the set of possible metamodel changes using predicates. Thirdly, we define in LP evolution operators and categorise them in primitive and compound operators.
4.1.1 Step 1 – Encoding MOF metamodels in LP

To reason about properties of metamodels and their evolution, a textual or a graphical representation is often not sufficient. Thus, we provide a more formal representation of metamodels in terms of sets and predicates. Our proposed approach consists in formalising MOF metamodels, by means of a LP formal language offering its own analysis capabilities. Therefore, as each model is an instance of the metamodel, a formalised model can be deduced. To be an efficient approach, the selected language must allow all the MOF metamodels features to be expressed and must offer an analysis capability to reason about evolution of MOF metamodels and their models. We use a formal language similar to the language proposed in Malgouyres (2006) for encoding UML models.

In this step we start by analysing MOF metamodels to identify evolutionary elements with their evolution parameters. We consider that an element is an evolutionary element if it can change (evolve) independently of its container, otherwise the element is considered as an evolution parameter in the context of its container.

As instance, In MOF metamodels ‘class’ element is linked with ‘property’ element by a composition link as shown in Figure 1. This link indicates that adding (respectively deleting or updating) a property is invoked by ‘class’ element. Thus, ‘class’ is the evolution context of ‘property’. In the other hand a property can be added (respectively deleted or updated) in a class without adding in advance a new class (respectively deleting or updating the same class). We deduce that ‘property’ is an evolution parameter in the ‘class’ context but ‘property’ is also an evolutionary element in the metamodel.

In this paper we focus only on the core metamodelling constructs that are interesting for models migration. We leave out constructs which cannot be instantiated in models as packages, annotations, derived features, and operations. Furthermore, metamodels differentiates the metamodel concrete features from the metamodel abstract features. This distinction is made because metamodel concrete features (e.g., classes) have their own instances in models whereas abstract features (e.g., generalisations) have not.

Because LP is based on facts and for considering modelling hierarchy, we define here a format to represent constructs called meta-facts (M2 level). The instantiation of a meta-fact is a fact (M1 level) which consists in giving constant values to the parameters of this meta-fact. The relationship between a meta-fact and a fact respects metamodelling principle. Every layer is an instance of the higher layer. Thus, facts allow representing instances of a model. A meta-fact is defined by its name, the number of its arguments and the meaning of each of its arguments. This definition of meta-fact can be represented as a meta meta-fact. Therefore, concrete elements in MOF are represented by meta meta-facts (M3 level) which are instantiated in meta-facts according to the metamodel. Thus, each metaclass in MOF metamodel can be translated on meta-fact by instantiating the meta meta-fact presented in equation (2) By instantiating this meta meta-fact we got a set of meta-facts, representing all metaclasses in MOF metamodels. In the other hand navigable relationships between metaclasses are represented by meta-facts instantiated from the meta meta-fact presented in equation (3). Moreover, some attributes can appear in models through other metaclasses specialising abstract classes. For example, attributes of NamedElement (e.g., name) are present in models through other metaclasses (e.g., class, property). Because NamedElement is abstract and does not have direct instances in model level. Thus, on facts presence of a NamedElement can be deduced from the presence of a class or a property. This relation between metaclasses represents
Using logic programming for adapting models to metamodel evolution

generalisation/specialisation relationship, which can be expressed by instantiating meta meta-facts in the rule presented in equation (4).

According to this method for encoding metamodels we have developed a set of clauses represented in Table 1, encoding MOF metamodels in two levels (M3 and M2) to which we have added additional functions used to facilitate manipulation of the metamodel and to access to different metamodel characteristics. For example, we define a function called ‘getName’ as presented in equation (5) to access name elements.

### Table 1 Extract from LP clauses encoding MOF metamodels

<table>
<thead>
<tr>
<th>Meta meta-fact</th>
<th>MetaClass_Name(Identifier, A1, ..., An).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta facts</td>
<td>namedElement(Id, Name, Visibility).</td>
</tr>
<tr>
<td></td>
<td>class(Id, Name, Visibility, IsAbstract).</td>
</tr>
<tr>
<td></td>
<td>feature(Id, Name, Visibility).</td>
</tr>
<tr>
<td></td>
<td>structuralFeature(Id, Name, Visibility, Type, Isunique, Lower, Upper, Isreadonly).</td>
</tr>
<tr>
<td></td>
<td>property(Id, Name, Visibility, Type, Isunique, Lower, Upper, Isreadonly).</td>
</tr>
<tr>
<td></td>
<td>A1, A2, ..., An : attributes</td>
</tr>
<tr>
<td>Meta meta-fact</td>
<td>Association_End_Name(id1, id2).</td>
</tr>
<tr>
<td>Meta facts</td>
<td>ownedAttribute(IdClass, IdProperty).</td>
</tr>
<tr>
<td></td>
<td>class(IdProperty, IdClass).</td>
</tr>
<tr>
<td></td>
<td>superClass(IdClass, IdClass).</td>
</tr>
<tr>
<td></td>
<td>memberEnd(Idassociation, IdProperty).</td>
</tr>
<tr>
<td></td>
<td>association(IdProperty, Idassociation)</td>
</tr>
<tr>
<td>Meta meta-fact</td>
<td>General_Class(Id, A1, ..., An) :- Special_Class(Id, B1, ..., Bm)</td>
</tr>
<tr>
<td>Meta facts</td>
<td>feature(Id, Name, Visibility) :- structuralFeature(Id,Name, Visibility, Type, Isunique, Lower, Upper, Isreadonly).</td>
</tr>
<tr>
<td></td>
<td>getName(id, Name) :- namedElement(Id, Name, _).</td>
</tr>
</tbody>
</table>

In order to automatically encoding in LP a metamodel or a model, a tool is developed which uses the XMI format of the considered metamodel (or a model). Analysing this XML file allows constructing a set of meta-facts and rules representing the metamodel. This tool is developed under Eclipse platform in Java and uses the Dom4j parser for analysing XML files (Dom4j, 2005). The overall metamodels encoding process used in this tool is shown in Figure 5.

We take as example a subset of Petri net metamodel (MM2) depicted in Figure 3(b) to show analysis steps of the metamodel. The analysed metamodel is encoded in XMI the listing in Figure 6 is the subset which includes definitions of some classes, associations and generalisation/specialisation relationships. We notice in this code that inherited
properties are not represented. Thus, to get the set of attributes of a meta-class we must consider also meta-generalisation between meta-classes.

Figure 5  Overview of metamodel encoding process

Figure 6  Listing of a subset of Petri net metamodel (MM2) in XMI format (see online version for colours)
The main steps of running the encoding process are:

1. Reading the metamodel with Dom4j tool (Dom4j, 2005) and recovery of metamodel root element (we note that Dom4j uses a reading tool called SAX XML).

2. Constructing for each metaclass the related meta-fact as follows:
   - Extracting the name of the meta-fact which corresponds to the name of the class.
   - Extracting arguments of the meta-fact which are constructed from attributes of the class that do not refer to an association.
   - Addition of meta-fact to the set of incomplete meta-facts. Those meta-facts are incomplete because inherited attributes of meta-classes are not considered. For example, the attribute ‘weight’ is not formalised in ‘Ptarc’ class and ‘Tparc’ class. In fact this attribute is inherited from ‘arc’ class.

After that, we use a specific method to extract meta-generalisation in order to calculate automatically complete meta-facts. For the previous example final meta-facts calculated by this tool are shown in Figure 7.

Figure 7 Petri net metamodel in LP formalism (see online version for colours)

4.1.2 Step 2 – Encoding metamodel changes in LP

We model differences between an original metamodel MM1 and an evolved version MM2 as a set of changes. These changes are of three kinds: additive changes, where the evolved metamodel contains an element that was not present in the original metamodel. Subtractive changes, where the evolved metamodel misses an element which was present in the original metamodel. Modification changes, where the evolved metamodel contains an element which corresponds to an element in the original metamodel but the value of a meta-feature in MM2 is different from its value in MM1.

For encoding these changes in LP formalism we adopt also a method respecting metamodelling principle. Thus, we propose the meta meta-fact presented in equation (6)
to represent changes. Where ‘Type’ and ‘Id’ indicate respectively the type and the identifier of the element subject of the change; \([P_1, …, P_n]\) is a set of parameters according to the change kind and meta element features. This meta meta-fact corresponds to M3 level. In M2 level, we define meta-facts according to possible metamodel changes described in the literature (Gruschko et al., 2007; Wachsmuth, 2007; Cicchetti, 2008; Hermannsdoerfer, 2010); some of these changes are represented in Table 2.

\[
\begin{align*}
\text{Change}_\text{Kind}(\text{Type}, \text{Id}, [P_1, …, P_n]) \\
\text{Change}_\text{Kind} \in \{\text{added}, \text{deleted}, \text{updated}\} \\
\text{Type} \in \{\text{class, property, association, supertype}\}
\end{align*}
\]

Table 2  \(\text{LP clauses encoding some metamodel changes.}\)

<table>
<thead>
<tr>
<th>Change meta fact</th>
<th>Change description</th>
</tr>
</thead>
<tbody>
<tr>
<td>added(class, Idc, [Name]).</td>
<td>Create a new class Name</td>
</tr>
<tr>
<td>added(class, Idc, [Name, Visibility, IsAbstract]).</td>
<td>Create a new class Name with specific values of visibility and IsAbstract.</td>
</tr>
<tr>
<td>added(property, Idp, [Idc, Name]).</td>
<td>Create a new property Name for the class Idc.</td>
</tr>
<tr>
<td>added(property, Idp, Idc, Name, IsReadOnly, IsIdentifier).</td>
<td>Create a new property Name for the class Idc with specific attributes IsReadOnly, IsIdentifier.</td>
</tr>
<tr>
<td>added(association, Ida, [Ids, Idc, Name, IsComposite , lower, Upper]).</td>
<td>Create a new association Name from class Ids to class Idc with specific attributes IsComposite, lower and upper values.</td>
</tr>
<tr>
<td>added(superType, Ids, [Idg])</td>
<td>Make class Idg super type of class Ids</td>
</tr>
<tr>
<td>deleted(class, Idc).</td>
<td>Drop class Idc</td>
</tr>
<tr>
<td>deleted(property, Idp).</td>
<td>Drop property Idp</td>
</tr>
<tr>
<td>deleted(association, Ida).</td>
<td>Drop association Ida</td>
</tr>
<tr>
<td>changed(class, Idc, [Name1, Name2]).</td>
<td>Rename class Idc from Name1 to Name 2</td>
</tr>
<tr>
<td>changed(association, ida, [lower1, lower2]).</td>
<td>Modify Lower bound of association ida from lower1 to lower2</td>
</tr>
</tbody>
</table>

4.1.3  \(\text{Step 3 – Encoding evolution operators}\)

This step permits to propose evolution operators which when applied on a metamodel version results a valid newer version. It is realised only once but is the core of our proposal. In this step, two tasks are performed:

- analysing MOF metamodels to identify and formalise authorised atomic evolution operators
- definition and formalisation of composite evolution operators applicable on MOF metamodels.

In the following we will define evolution operators applicable only on a set of evolutionary elements. Since evolutionary elements are already encoded using meta-facts described in the first step. The aim in this stage is to extract from MOF metamodels the definition of operators that are implicitly authorised and applicable on these elements and
resulting a valid metamodel, because evolution operators are not explicitly specified within a metamodel.

Operators originate from the literature of metamodels evolution. Wachsmuth (2007) first proposes a set of operators according to the preservation of metamodel expressiveness and existing models. Gruschko et al. (2007) envision a difference-based approach and therefore classify all primitive changes according to their impact on existing models. Cicchetti (2008) lists a set of composite changes used in a difference-based approach. Herrmannsdoerfer et al. (2011) propose a catalogue of 61 operators for the coupled evolution of metamodels and models. These coupled operators evolve a metamodel and are able to migrate automatically existing models. The catalogue covers operators presented in previous works and also operators which have proven useful in a number of case studies performed by the authors. Proposed catalogue is based on EMOF metamodeling formalism (OMG, 2015b) and contains primitive operators as well as complex operators. We have found that this catalogue is very rich. In our proposal, we use an analogue catalogue but we focus firstly on encoding operators in LP formalism in order to allow reasoning, secondly on organising the operators in a way where the operator evolution and co-evolution parts will be clearly defined in order to facilitate the library extension (i.e., evolution) and thirdly evolution operators will be used in posterior that means after metamodel evolution. Therefore, we envision to get understandable definition of operators and easy to be reused.

Firstly, we propose a set of primitive operators. Primitive operator performs an atomic metamodel evolution step that cannot be further subdivided. Each atomic evolution operator is structured as follows:

- operator description: this section presents informally the role and motivation behind the operator
- operator definition: in this rubric we propose a formal definition of the operator using LP formalism
- operator prior condition required to instantiate an operator
- operator post condition resulted after executing an instantiated operator
- operator evolution action used only to test the validity of evolution scenario
- operator model adaptation action used to generate model migration scenario.

Before starting operator’s definition, we present some notation to be used in the following:

- TypeOf(E): return le type de E
- OldMMEElements: the set of instances of evolutionary elements defined in the old version of the metamodel
- NewMMEElements: the set of instances of evolutionary elements defined in the new version of the metamodel
- Ownedmembers(E): the set of elements that E is a container
- Add-collection(S, E): add to the set S the element E.
As instance, the primitive operator ‘Create property’ is structured as follows:

**Operator description**

The operator ‘Create property’ allows adding a new element ‘property’ to an existing class in a metamodel.

**Operator definition**

create_property(Idproperty, Idclass, visibility, type, Isunique, lower, upper, Isreadonly, Iscomposite, IsDerived, default).

**Prior-condition**

If (Idclass ∈ oldMMElements) then

If Typeof(Idclass) = class and

Idproperty ∉ ownedmembers (Idclass) then

Add-collection (ownedmembers(Idclass), Idproperty)

**Post-condition**

property(Idproperty) ∈ ownedmembers(Idclass).

property (ldproperty) ∈ NewMMElements

**Operator evolution action**

Add an element property to the metamodel with specific features. The property has as a container ‘Idclass’ : property(Idproperty, Idclass, visibility, type, Isunique, lower, upper, Isreadonly, Iscomposite, IsDerived, default)

**Operator model adaptation**

If (lowerValueof(Idproperty) = 0) then the property is optional, and no adaptation is needed.

If (lowerValueof(Idproperty) > 0 ) and defaultvalueof(Idproperty) <> null then the property is instantiated automatically, and model adaptation is automatic.

If (lowerValueof(Idproperty) > 0 ) and defaultvalueof(Idproperty) = null then

if typeof(Idproperty) ∈ {String, Boolean, Float, Enumeration } then values options are displayed, user selects a value to initialise the property. Model adaptation is user driven.

if typeof(Idproperty) = class and defaultValueof(Idproperty) = null then creation of class instance cannot be done automatically. Adaptation of the model is manual.

Primitive operators considering principal evolutionary elements are given in Table 3. They are represented by meta-facts each one with a set of formal parameters.

Composite operators can be decomposed into a sequence of primitive operators which have the same effect at the metamodel level but typically not at the model level. Composite or compound changes have been already considered in previous works like (Herrmannsdoerfer et al., 2008; Cicchetti, 2008). Therefore, in our proposal we formally define them using LP formalism. We note that further intermediate functions are represented in these rules representing conditions needed to execute operations.

We will define only a few composite operators in this paper shown in Table 4. As instance, we consider extract super class operator where a class is generalised in a hierarchy by adding a new general class and two references to their subclasses. Or, feature pull up operator which can be decomposed into feature deletion in the subclasses followed by a feature creation in the parent class and so on. Complete set of primitive and compound operators results a knowledge base which will be used by the inference engine to reconstruct evolution scenarios.
Using logic programming for adapting models to metamodel evolution

Table 3  Extract from primitive evolution operators in LP clauses

<table>
<thead>
<tr>
<th>Primitive operators definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_class(Name, Visibility, IsAbstract)</td>
</tr>
<tr>
<td>delete_class(Name)</td>
</tr>
<tr>
<td>create_property (Idproperty, ldclass, visibility, type, Isunique, lower, upper, Isreadonly, Iscomposite, is derived, default)</td>
</tr>
<tr>
<td>create_asso(Idass,Name_a, ldclass_s, ldclass_T, lower, upper, Iscomposite, Isopposite)</td>
</tr>
<tr>
<td>delete_property(Idproperty, ldclass_c)</td>
</tr>
<tr>
<td>rename_class(ldclassc1, Name_c)</td>
</tr>
<tr>
<td>rename_property(Idproperty, Name_p)</td>
</tr>
<tr>
<td>make_abstract(Idclass)</td>
</tr>
<tr>
<td>drop_abstract(Idclass)</td>
</tr>
<tr>
<td>add_super(Idclass_s, ldclass_G)</td>
</tr>
<tr>
<td>drop_super(Idclass_s, ldclass_G)</td>
</tr>
<tr>
<td>make_composite(Idass, ldclass_s)</td>
</tr>
<tr>
<td>drop_composite(Idass, ldclass_s)</td>
</tr>
<tr>
<td>make_opposite(Idass, ldclass_s, ldop, ldop_s)</td>
</tr>
<tr>
<td>drop_opposite(Idass, ldclass_s, ldop, ldop_s)</td>
</tr>
<tr>
<td>generalize_lower(Idass, ldclass_s, L)</td>
</tr>
<tr>
<td>specialize_lower(Idass, ldclass_s, L)</td>
</tr>
<tr>
<td>generalize_upper(Idass, ldclass_s, U)</td>
</tr>
<tr>
<td>specialize_upper(Idass, ldclass_s, U)</td>
</tr>
<tr>
<td>make_identifier(Idproperty, ldclass)</td>
</tr>
<tr>
<td>drop_identifier(Idproperty, ldclass)</td>
</tr>
</tbody>
</table>

Table 4  Examples of composite evolution operators in LP clauses

<table>
<thead>
<tr>
<th>Composite operators definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull_up_feature(Idclass, ldproperty) :- typeof(ldproperty, property), findall(Cs, super(Idclass, Cs), Ls), delete_property(ldproperty, Ls), create_property(Idproperty, ldclass).</td>
</tr>
<tr>
<td>Pull_up_feature(Idclass, ldproperty) :- typeof(ldproperty, association), findall(Cs, super(Idclass, Cs), Ls), getName(ldproperty, Name_a), getType(ldproperty, Type), getLower(ldproperty, L), getUpper(ldproperty, U), getComposite(ldproperty, Comp), delete_property(ldproperty, Ls), create_asso(Name_a, ldclass, Type), specialize_lower(ldproperty, ldclass, L), generalize_upper(ldproperty, ldclass, U).</td>
</tr>
<tr>
<td>extract_super_class(Name_c, [ldclass1, ..., ldclassk], [ldproperty1, ..., ldpropertyj]) :- create_class(Name_c,public,true), getId(ldclass, Name_c), add_super(ldclass1, ldclass), ..., add_super(ldclassk, ldclass), pull_up_feature(ldclass, ldproperty1), ..., pull_up_feature(ldclass, ldpropertyj).</td>
</tr>
</tbody>
</table>

4.2 Detection of changes

Detection of differences between models is essential to model development and management practices. Thus, evolution from one metamodel version to the next can be described by a sequence of changes. Understanding how metamodels evolve or
discovering changes that have been performed on a metamodel is a key requirement before undertaking any migration operation on models to co-evolve them. In fact, we distinguish two ways for discovering changes (Cicchetti et al., 2007). One way, changes applied to the metamodel have to be detected in posterior using state-based model comparison approaches with either generic model comparison algorithms or language-specific comparison algorithms. Another way to acquire the set of changes is to use operation recording; that is, the execution of evolution operations is tracked within the modelling environment while they are performed. The later solution can be used only if metamodels are evolved by the same tool in order to capture the evolution tracks. Our approach uses the first method. So, for detecting the set of changes performed to the older version of the metamodel in order to produce the new one, we use generic algorithm. Whereas current generic approaches only support detecting atomic changes, some language-specific approaches allow detecting composite changes also; but only for one specific modelling language (Herrmannsdoerfer et al., 2008).

Primitive differences between metamodels versions are classified in three basic categories: additions, deletions, and updates (i.e., modifications) of metamodel elements (Cicchetti et al., 2007). These differences represent atomic changes (i.e., elementary).

Detected differences are represented as elementary changes specifying fine-grained changes that can be performed in the course of metamodel evolution. There are a number of primitive metamodel changes like create element, rename element, delete element, and so on. One or more of such primitive changes compose a specific metamodel evolution.

There are several tools for matching and differencing models. In our solution, computation of differences between metamodel versions is performed with Eclipse plug-in EMF Compare (EMF, 2015b). EMF Compare is a prominent representative of model comparison tools in the Eclipse ecosystem. This tool provides algorithms to calculate the delta between two versions of a model and visualises them using tree representation. EMF Compare engine represents the list of differences as atomic changes. Each one includes essentially information about the type of change, the element subject of change and also some evolution parameters. Additionally, within each elementary change are provided two lists that are prior and post list. The first one contains a set of changes that must be performed before doing the change under hand named ‘requires list’. This set constitutes the prior condition to execute a specific change. The second list named ‘required by list’ which groups changes that require performing considered change before them. EMF Compare is capable of detecting the following types of atomic changes: add, delete, update and move.

The set of detected differences is called delta model. This later is translated in LP formalism using the developed tool. Thus, the delta model is translated in XMI format and passed as input to the encoding tool, specific methods are used to extract automatically the change kind and its parameters and translate it into facts according to the description defined in equation (6). This result is used in the next phase to reconstruct the evolution scenario. More details on this step are given in the following section.

Figure 8 shows the resulted difference model calculated with EMF Compare (EMF, 2015b) using the two versions of Petri net metamodel MM1 [Figure 3(a)] and MM2 [Figure 3(b)]. This difference model is then translated in LP by the developed tool as explained above which results facts depicted in Figure 9.
4.3 Reconstruction of evolution scenario

In this section, we discuss the reconstruction of evolution scenario done in two steps. In the first one, we reconstruct a correctly ordered evolution scenario from the delta model, by converting facts of changes to primitive evolution operations. In the second step, we use the knowledge base of operators to reconstruct composite operations by means of an inference engine.

4.3.1 Step 1 – Reconstructing primitive evolution operators

In this step atomic changes calculated in the delta model and coded with facts are transformed into sequence of primitive operations called a primitive evolution scenario which describe the evolution of a metamodel. We note that an operation is an instantiation of operator parameters with actual arguments.

Firstly, we convert differences in atomic operations conforming to operators definitions defined in the library of operators. For this purpose, we use an LP inference
engine with a base of rules in order to obtain from each atomic change in delta model the corresponding primitive metamodel evolution operator. Examples of inference rules used to convert changes into corresponding operations are presented in the following listing.

In fact, evolution scenario is stocked as a base of facts, each one represent a primitive evolution operator. After that we reorganise evolution operations because detected differences are not always presented in the correct order which invalidate evolution scenario. Thus, we used the prior and post conditions to define dependency between operations which allows reorganising the primitive evolution scenario. Finally this step checks whether the evolution scenario is valid, by applying the resulted scenario to the old metamodel version, if it results the new metamodel version then primitive evolution scenario is well ordered else the reorganisation process is repeated.

(see online version for colours)

In the previous example, we use Prolog inference engine to get the primitive operations from the set of changes; a part of them is in the following listing:

(see online version for colours)

```
4.3.2 Step2 – Reconstructing composite evolution operators

Granularity of metamodel evolution changes is not always appropriate. Often, intent of the changes may be expressed on a higher level. Thus, a set of atomic changes can have together the intent of a composite change. For example, generation of a superclass ‘sc’ of two classes ‘c1’ and ‘c2’ without common features can be done through successive applications of a list of primitive changes, such as creating a class ‘sc’ (i.e., added(class, Idsc, [sc])), adding a reference from ‘c1’ to ‘sc’ (i.e., added(superType, Idc1, [Idsc])), and adding a reference from ‘c2’ to ‘sc’ (i.e., added(superType, Idc2, [Idsc])).

In this proposal we applied an intelligent reasoning to deduce compound operations of evolution performed by metamodel designer to evolve the metamodel. Reorganising the set of these operations results an evolution scenario.
```

```
In this step we reconstruct composite operation from valid primitive evolution scenario. We have already specified the definition of compound operators in a knowledge base using rules. Facts contained in the evolution scenario are passed as input to an inference engine which will use the knowledge base of operators to deduce composite operations. Once a composite operation is inferred, the sequence of associated primitive operations is merged by adding the composite operation to the base of facts (i.e., assert fact) and consequently to the scenario and removing facts of corresponding primitive operations from the base (i.e., retract fact). This results in a new evolution scenario which will be validated by executing it on the old version of the metamodel. Possible reorganisation may be done to authorise inferring composite operations. This process is repeated until no deduction can be further made. Final valid evolution scenario will contain primitive operations as well as composite ones.

The base of facts representing atomic changes for Petri net metamodel evolution constitutes primitive evolution scenario which will be used by the Prolog inference engine with the knowledge base defining composite operators in order to deduce eventual composite operations.

As instance the composite operation presented in equation (7) will be inferred. This fact indicates that a new class with the identifier ‘idc26’ is created and both classes with ‘idc24’ and ‘idc25’ identifiers get ‘idc26’ class as their superclass.

Thus, we can see in equation (7) that our composite operation ‘extract superclass’ contains three primitive changes (one ‘create class’ and two ‘create superclass’).

\[
\text{extract superclass(idc26, [idc24, idc25])}
\]  

(7)

4.4 Generation of migration scenario

As previously explained, the library regroups evolution operators associated with information about how to migrate models. Evolution operators have different impacts on models conformity (Gruschko et al., 2007). We define four categories of operators according to this impact. An operator is free impact when change performed by applying this operator is not breaking. The operator is automatic, if the change resulted by applying an operator is breaking and resolvable and for this category the operator is reusable and every time migration procedure will be automatically executed. Whereas, operators are considered user driven or manual if changes are breaking and not resolvable. Operators are user driven where migration of models cannot be completely automated and need more information from the model designer however when model migration cannot be defined automatically and adaptation solution must be provided manually by model designer the operator is manual.

Migration procedure varies according to the operator’s category. It is implemented in Java. It contains some tests of operation parameters to determine which category impact is the case and thus find eventual transformation to apply on the model. For reusable operators, migration procedure is automatically obtained by instantiating parameters. However, for user driven operators, we have defined migration procedures by explicitly specifying some alternatives solutions to assist user. Consequently, these solutions allow users to customise migration of their models. Thus, the complete migration scenario is composed by a sequence of invoked methods with eventual parts where the transformation is specified manually by model designer to complete model co-evolution process.
4.5 Migration

This phase takes as input an instance model conforming to the initial metamodel. This model is also called user model. In order to transform the model to conform to the newer metamodel version, migration scenario will be applied to the input model. Some parts of the scenario will be automatically executed. In other parts the system assists users in solving the changes by presenting alternative solutions specified in the migration scenario. User selects through a graphic user interface among different alternatives for each affected element, and provides any additional information required by the selected alternative. For instance, if the metamodel evolution includes the addition of a new property, solution alternatives can include the initialisation of this property with a predefined value or with a derived value based on other already existing properties. Additionally, users can provide extra information to complete the change on the model if necessary. For instance, if the new property must be initialised, the user must also be requested for the initial value. Thus, the migration is performed based on a set of assistance options.

Figure 10 A sample Petri net metamodel (see online version for colours)

As instance, from the validated evolution scenario, we take migration procedures defined within operators and execute them on a specific Petri net represented by its equivalent model M1 compliant to MM1 as depicted in Figure 10. To co-evolve M1 with the newer
metamodel version MM2, new instance objects are created according to new metaclasses and also new associations and attributes. Furthermore some associations will be dropped. During this step user will be asked about values of mark property of every token associated with place objects as well as values of weight property of different created arcs. As instance P1, P2, P3 and P4 places will be marked with values (4, 2, 1, 0) respectively and the different arcs will be initialised with default value which is 1. The result of adapting the previous model to the new version of the metamodel is depicted in Figure 11.

5 Conclusions

Automating the co-evolution of models and metamodels is a challenging task. In this paper we have proposed an alternative solution to this problem. Thus, our proposal illustrated a hybrid approach to guide the user in solving co-evolution issues. It takes advantages from both state-based and operator-based approaches adding a new dimension to deal with this problem which is using an intelligent reasoning. It resolves around three axes, namely changes detection, reconstruction of evolution and migration scenario and migration of models. Our solution is independent from any modelling language and it is easily adapted to various modelling environment. It consists of using a library of coupled evolution operators, but unlike existing approaches, changes requiring user intervention are also integrated in the library with specific alternatives solutions to assist automatically users during migration activity. Moreover, the benefits of this approach are numerous, notably encoding metamodels in LP clauses and the formalisation of evolution operators in order to allow reasoning and to facilitate eventual extension of the operator’s library. The intelligent reasoning used in reconstructing evolution scenario constitutes the core of co-evolution process. It ensures a real separation between the metamodel evolution and model co-evolution and consequently between metamodel designer and model designer. Using intelligent logic mechanism to infer compound evolution operators and evolution scenario increases effectiveness of our proposal and makes our solution distinguishable from existing works.

Evaluation is not linearly made on all phases of the approach. However we have tried to test the core of our approach which allowed us to evaluate the encoding process on which repose the four phases of the approach. Furthermore, we have tested the reconstruction mechanism of the evolution on which repose the definition of the needed migration. Throughout the evaluation of our approach we have used Petri net metamodel evolution scenario. The preliminary validation results are promising and prove that the presented approach is feasible. We deduce that the proposed encoding process in LP as well as the specification of evolution operators is satisfactory and improve the comprehension of the metamodels evolution for the model designer.

We have as perspective to complete implementation of the catalogue of operators in this prototype and we will conduct real case studies for validation. In addition, we think about extending our approach to support the co-evolution of not only instance models, but also other artefacts depending on the same metamodel like transformation rules. We will study also possibilities to extend our solution in order to support representation of semantic in models and preserving semantics within the migration process as introduced in Cicchetti and Ciccozzi (2013).
References


