JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

On the Impact Significance of Metamodel Evolution in MDE

Ludovico Iovino[a] Alfonso Pierantonio[a] Ivano Malavolta[a]

a. Department of Computer Science, University of L'Aquila, Italy
http://www.di.univaq.it

Abstract Harnessing metamodels to engineer application domains is at the core of Model-Driven Engineering. A large number of artifacts pursuing a common scope are defined starting from metamodels which represent the nucleus of an ecosystem. Analogously to any software artifact, metamodels are equally prone to evolution. However, changing a metamodel might affect the components of the ecosystem. In fact, when a metamodel undergoes modifications, the related artifacts might require to be consistently adapted in order to recovery their validity. This is an intrinsically difficult process. It requires different techniques for each specific kind of artifact and can easily lead to inconsistencies and irremediable information erosion, if based on spontaneous and individual skills. This paper discusses the problem of identifying, predicting and evaluating the significance of the metamodel change impact over the existing artifacts. The approach is agnostic of the adaptation technique and formalizes the whole ecosystem and the relatedness of the involved artifacts in terms of megamodels. This allows developers *i)* to establish relationships between the metamodel and its related artifacts, and *ii)* to automatically identify those elements within the various artifacts affected by the metamodel changes. The approach can be considered as preparatory to any systematic adaptation process.

Keywords Model driven engineering, coupled-evolution, metamodeling

1 Introduction

In recent years, Model-Driven Engineering [31] (MDE) has taken a leading role in advancing a new paradigm shift in software development. Domain metamodels are at the core of this discipline, as most of the constituent artifacts and components are based on them, including models and transformations. However, the entities which are defined upon domain metamodels are numerous and include also syntax-directed and diagrammatic editors, model differencing and analysis tools, just to mention a few (see Figure 3).

Analogously to any other software artifact, domain metamodels are living entities too. Indeed, they are subject to evolutionary pressures as soon as new insights about the domain emerge and require to be consistently reflected in the metamodel formalization [17]. Consequently, new requirements may need to be accommodated in the metamodel and existing

ones may require to be refined, or even amended. Unfortunately, whenever a metamodel undergoes modifications, all the related artifacts must be accordingly *adapted* in order to remain valid. Naturally, the modeler and the implementors can always adapt the models, the transformations and any other related tool by inspecting the artifacts, detecting the necessary refactorings, and finally applying them with manual operations. However, carrying on this activity without specialized tools and techniques presents intrinsic difficulties. The intricacy of metamodel changes on one hand, but also the size and diversity of the artifacts on the other hand, can rapidly affect the accuracy and precision of the adaptation [4]. In fact, if the adaptation is based on spontaneous and individual skills and pursued without any automated support, it can easily give place to inconsistencies and lead to irremediable information erosion [36].

In this paper, we discuss the problem of identifying, predicting and evaluating the significance of the domain metamodel change impact over the existing artifacts. Documenting and formalizing the relatedness of all the involved entities is preparatory to any systematic adaptation process. To this end, we propose an approach that is agnostic of the adaptation technique and permits *i)* to establish relationships between the domain metamodel[1] and its related artifacts, and *ii)* to *automatically* identify those elements within the various artifacts affected by the metamodel changes. The approach focusses on the exploration and definition of the various relationships that may exist between the involved modeling artifacts and the modeling language. The approach contributes in several ways:

– the interdependencies not always are explicit and easily observable, thus documenting them in a rigorous way can provide an adaptation developer with information and insight normally obscured by the intricacy of the modeling structures or even by the code;

– it eases the assessment of the impact significance of metamodel changes by generating from the above specification documents which detect and highlight the affected modeling elements.

In essence, harnessing the opportunity of using formal documentation and automated support for the detection and analysis of the adaptation requirements is crucial for reducing the number of false positives. The approach is based on megamodeling, a promising technique for representing and managing relationships that may exist between a set of modeling artifacts, for this reason this technique has been employed to represent the relations between artifacts and domain metamodel; therefore, it is a fundamental building block for possible approaches to automatically derive either the corrupted or involved model elements after a change in the domain metamodel occurred.

Structure The paper is organized as follows. In Section 2 motivation and exemplars of coupled evolution scenarios are illustrated and the steps involved in coupled evolution are identified. In Section 3 we discuss how the identified relations in Section 2 can be formalized in order to automate the identification of impacted elements. Then, Section 4 presents how conformance has been formalized in the context of our approach. Section 5 describes how we exploit megamodeling for managing the coupled evolution of modeling artifacts. A scenario in which we apply our approach and its usefulness in practice are provided in Section 6. Finally, Section 7 discusses related work, and Section 8 provides conclusions and future work directions.

[1] In this work we will use the term *domain metamodel* wherever we refer to the metamodel describing the problem domain; the terms *metamodel*, *domain model*, and *modeling language* can be seen as synonyms of domain metamodel.

2 Modeling Artifacts Live in Ecosystems

In MDE domain metamodels are core ingredients which rarely exist in isolation, i.e., a domain metamodel can be considered – together with the modeling artifacts that depend on it – as part of an ecosystem, i.e., a set of inter-related entities which are used together for a common objective[2]. With reference to Figure 1, changing a metamodel triggers a ripple effect

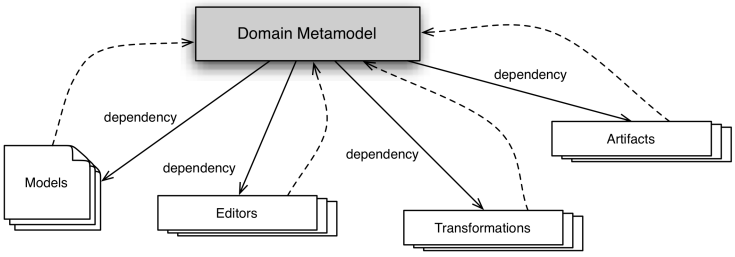


Figure 1 – Generic modeling artifacts ecosystem

which might affect the validity of the artifacts according to the dependencies existing among them. In the sequel of the section, we analyze the nature of such *dependencies* in order to better understand the consequences of the metamodel evolution over the related artifacts [33]. In order to analyze these interdependencies we consider the classical workflow for using the domain metamodel which consists of the following stages:

1. *Domain Metamodel Development*. Domain experts and developers work together to iteratively create an initial version of the domain metamodel. The process alternates between domain metamodel changes and some form of validation until the domain metamodel accuracy, expressiveness, and coverage are satisfactory [10].

2. *Domain Metamodel Usage*. Models are instantiated according to the metamodel definition. At this stage, the domain metamodel is more quiescent and is eventually subject to only minor changes. However, relevant changes are still possible and may be eventually recorded to be later analyzed in the next step.

3. *Domain Metamodel Evolution/Maintenance*. Once a significant number of relevant changes have been collected, the designer must assess the impact significance of the requested changes and eventually proceed with a metamodel refactoring.

The last step is the most critical as it can invalidate the ecosystem. Hence, our approach advocates the necessity of assessing the significance of the metamodel refactorings in order to evaluate the effort for restoring the ecosystem consistency. Designing the right adaptation for each kind of artifact is a difficult task and is traditionally not univocal [12]. Thus, a rigorous discipline is essential because if this problem is inaccurately handled, can increasingly render the domain metamodel resilient to variations. A typical example is provided by the GMF editors which respond to domain metamodel changes with difficulty, making the metamodel locked in [13]. Fundamentally, the adaptation process can be considered as a *three-steps process*:

[2] A more complete definition of software ecosystem says that it consists of the set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solution [8].

1. *Relations definition*. The correspondences between the domain metamodel and the artifacts are identified. Intuitively, those relations can be considered as dependencies between artifacts, and they play a role similar to that of tracing information between source and target models of a model-to-model transformation.
2. *Change impact detection*. The relations defined in the previous step are considered to assess the metamodel change impact on the related artifacts.
3. *Adaptation*. The information in the previous step are used to devise proper adaptation actions on the (possibly corrupted) artifacts. This step can employ different adaptation procedure, depending on the types of artifacts to be adapted.

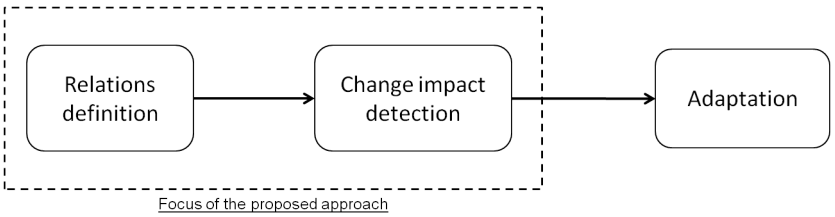


Figure 2 – Generic adaptation activities

In current practices these steps are traditionally not distinguished, so that the impact assessment and the adaptation semantics are blurred. The focus of this work is on the first and second steps described in Figure 2, with the main requirement of being agnostic of the specific solutions that can be applied in the third step.

Dependencies emerge at different stages during the domain metamodel life-cycle, and with different degrees of causality depending on the nature of the considered artifact. For instance, by referring to Figure 3, there may be a model *transformation* (endogenous, in this specific case) that takes as input a model *m1* and produces a model *m2*, both conforming to the domain metamodel; also a *graphical* or *textual tool* to describe models or other kinds of artifacts may be utilized, and all of them are strictly related to the domain metamodel.

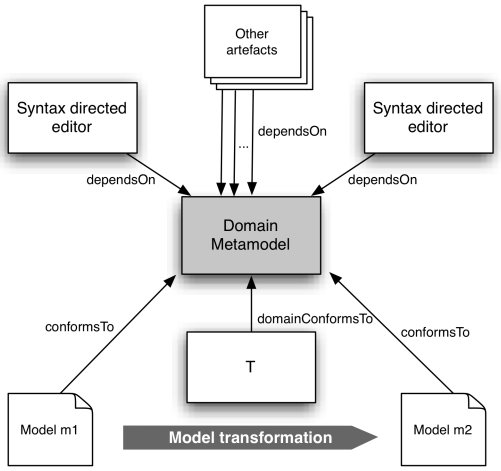


Figure 3 – Overview of the artifacts relation with the domain metamodel

Being more precise, metamodels may evolve in different ways: some changes may be additive and independent from the other elements, thus requiring no or little co-changes. How-

ever, in other cases metamodel manipulations introduce incompatibilities and inconsistencies which can not be always easily (and automatically) resolved. According to our experience and to the literature (e.g., [5, 9, 27, 26, 14]), at least the following relations are involved in the metamodel co-evolution problem:

- *conformsTo:* it holds between a model and a metamodel, it can be considered similar to a typing relation. A model *conforms to* a metamodel, when the metamodel specifies every concept used in the model definition, and the models uses the metamodel concepts according to the rules specified by the metamodel [5];
- *domainConformsTo*: it is the relation between the definition of a transformation and the metamodels it operates on [26]. For instance, a sample domain conformance constraint might state that the source elements of every transformation rule must correspond to a metaclass in the source metamodel [27] and same for target metamodel elements;
- *dependsOn*: it is a generic and likely the most complex relation, since it occurs between a metamodel and modeling artifacts, which do not have a direct and a well-established dependence with the metamodel elements. For instance, in case of EMFText [15] syntax specification, it does not refer directly to the elements specified in the metamodel, even though some form of consistency has to be maintained in order to not obtain EMFText editors with errors at runtime.

Clearly, the conformance and the domain conformance relation are contained in the dependence relation. To illustrate the relations described above, in the remaining of this section we will consider some cases about the evolution of a simple Petri net metamodel from the version shown in Figure 4 to the evolved version in Figure 5.

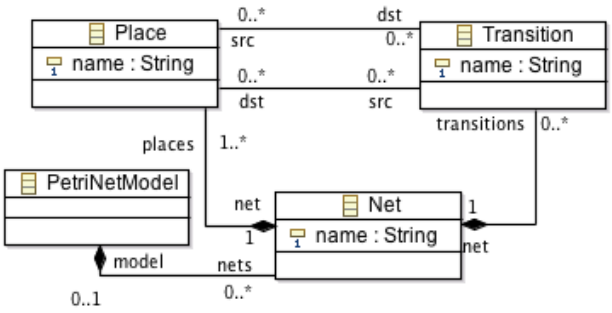


Figure 4 – PetriNet Initial metamodel version

In the evolved version of the metamodel the metaclasses *Arc*, *PlaceToTransition*, and *TransitionToPlace* have been added, and other changes have been executed like the merging of the references *places* and *transitions* into the new *elements*, and the renaming of the metaclass *Net* as *PetriNet*. Due to these relations described above, such modifications can affect those existing artifacts that are coupled with the Petri net metamodel, as discussed in the following sections.

2.1 Metamodel/model Co-evolution

The conformance relation *conformsTo* establishes the *typing* – and consequently the validity – of a model with respect to a metamodel. It can be considered analogous to the typing relation

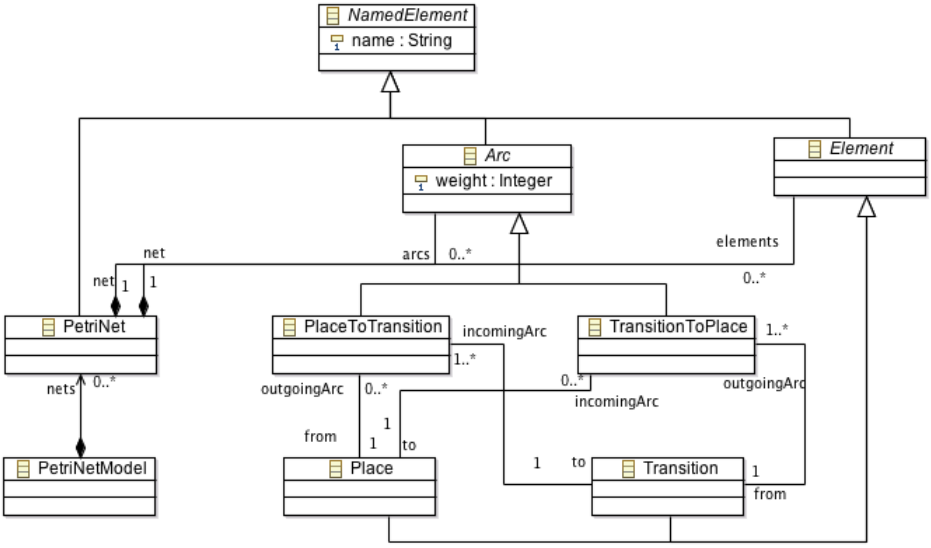


Figure 5 – PetriNet evolved metamodel version

between an object and its class. Changes in the metamodel might compromise the validity of the existing models in several ways depending on such a relation and their validity must be re-established according to the conformance relation. In this scenario, metamodel changes can be classified according to their corrupting effects [36]:

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

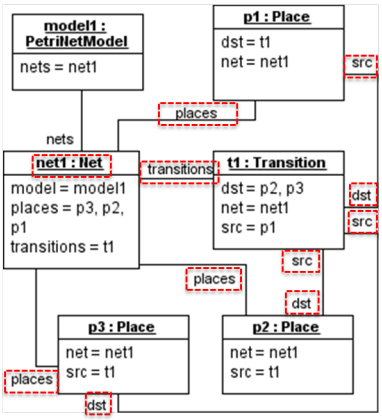


Figure 6 – PetriNet model affected by the simple metamodel evolution

For instance, Figure 6 shows a simple model conforming to the initial version of the PetriNet metamodel in Figure 4, whose validity is compromised by the changes performed on the metamodel to obtain the final version in Figure 5. In fact, the conformance relation is broken because of the modeling elements highlighted in Figure 6 with dotted lines. In particular, the element *net1* has to be adapted since the metaclass *Net* no longer exists. Moreover, the *places* and *transitions* references have to be changed since they have been merged into the new *elements* reference. Finally, the references *src* and *dst* have been replaced by some added class references.

2.2 Metamodel/transformation Co-evolution

Domain metamodel changes can compromise model transformations as well. In particular, inconsistencies arise when elements in the considered transformation no longer satisfy the *domain conformance* relation mentioned above. For instance, when a concept is removed from a domain metamodel, existing transformations that use the removed concept are no longer domain conformant. The inconsistencies with the domain metamodel are manifested at compile time and countermeasures must be adopted. In this respect, in the case of metamodel/transformation co-evolution, metamodel manipulations can be classified[3] as follows [24]:

- *fully automated:* changes that affect existing transformations which can be automatically migrated without user intervention;
- *partially automated:* changes are modifications that affect existing transformations which can be adapted automatically even though some manual fine-tuning is required to complete the adaptation;
- *fully semantic:* changes are those modifications which affect transformations which cannot be automatically migrated, and the user has to completely define the adaptation.

To better comprehend such a classification, let us consider the ATL [20] transformation shown in Listing 1. It is an excerpt of one of the transformations available in the ATL Zoo[4]. The transformation is designed to generate PNML[5] models starting from Petri net models. Also such a transformation is affected by the domain metamodel changes in Figure 4. For instance, in lines 4,9,16, there are references to the *Net* element, which does not exist in the new version of the domain metamodel. Moreover, in line 21 the references *places* and *transitions* are used, even though they have been merged into the new reference *elements*. Finally, the use of the references *src* and *dst* in lines 40 and 50 must be adapted. Indeed the transformation, when executed, gives errors at runtime and the developer has to find and fix the inconsistencies.

Listing 1 – Fragment of the PetriNet2PNML ATL transformation

```
1 module PetriNet2PNML;
2 create OUT : PNML from IN : PetriNetMM0;
3 ...
4 helper context PetriNetMM0!Net def: totPlace() : String =
5  self.places->select(e |
6 e.oclIsTypeOf(PetriNetMM0!Place)
7  ).size().toString();
8
9 helper context PetriNetMM0!Net def: totTransition() : String =
10  self.transitions->select(e |
```

[3] For the sake of simplicity, those changes in the domain metamodel which do not affect the existing transformations are not considered in the present discussion.

[4] www.eclipse.org/m2m/atl/atlTransformations/

[5] PetriNet Markup Language

```
e.oclIsTypeOf(PetriNetMM0!Transition)
 ).size().toString();
...
rule Net {
 from
s : PetriNetMM0!Net
 to
t : PNML!NetElement (
  name <- name,
  document <- thisModule.document,
  contents <- s.places.union(s.transitions),
  type <- type-uri,
  id <- s.name
),
 ...
label : PNML!Label (
  text <- s.name+' tot places:'+s.totPlace()+' and tot transitions:'+s.totTransition()
),
 ...
 do {
thisModule.document.nets <- t;
 }
}
rule Place {
 from
s : PetriNetMM0!Place
 to
t : PNML!Place (
  name <- name,
  id <- s.name + '-src:' + s.src.size().toString() + '-dst:' + s.dst.size().toString()
),
 ...
}
rule Transitions {
 from
s : PetriNetMM0!Transition
 to
t : PNML!Transition (
  name <- name,
  id <- s.name + '-dst:' + s.dst.size().toString()
),
 ...
}
```

2.3 Metamodel/Concrete Syntax Specification Co-evolution

The concrete syntax of a modeling language is usually given by linking the abstract syntax given in a metamodel with a (textual or diagrammatic) vocabulary. Thus, changes within a domain metamodel can also affect to different extents the concrete syntax specification by turning for instance existing mappings into dangling references. An example of notation for specifying the concrete syntax (and its related mappings) is EMFText [15]. For example, by referring to Figure 7, EMFText allows the user to define a textual syntax specification for Petri net models.

Starting from an EMFText specification, a number of supporting tools can be generated, like a parser (text-to-model) and pretty-printer (model-to-text) for domain models sentences, and a dedicated modeling environment for the domain metamodel, analogously to what happens with meta-environments for language development (e.g., see [23]).

In particular, the EMFText engine allows the user to build model-to-text and text-to-model transformations to serialize models and to parse sentences of the considered DSML, respectively. These operations, called injection and extraction, are bidirectional and can be executed by means of model-to-code transformations. For each structural element defined in the domain metamodel, its corresponding syntactical constructs are defined in EMFText.

EMFText syntax specifications are expressed as a set of rules written by following a

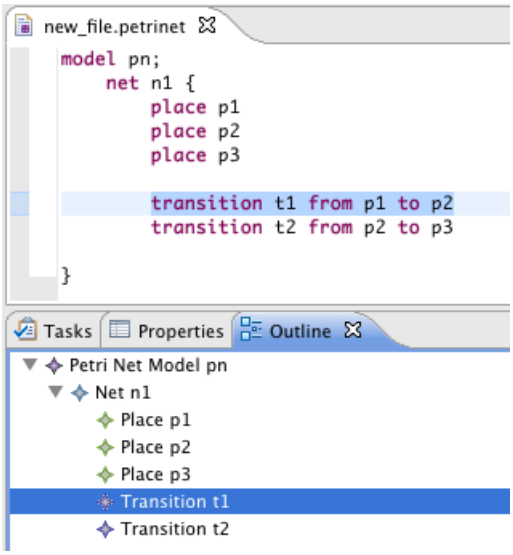


Figure 7 – A simple Petri net textual model using EMFText concrete syntax specification

grammar-like style. Clearly, changes in the domain metamodel can affect an existing EMFText syntax specification. Thus, similarly to what holds for ATL transformations (see Section 2.2), a *dependsOn* relation exists between the domain metamodel and its corresponding concrete syntax specifications. Moreover, in order to precisely estimate what is the impact of a domain metamodel evolution, each part of the syntax specification impacted by the changes in the domain metamodel must be identified. For example, Figure 8 shows a fragment of the EMFText syntax specification for Petri nets, impacted by the domain metamodel evolution. Highlighted elements are the syntax errors after changes within the domain metamodel have been performed. For instance, the *Net* metaclass has been renamed to *PetriNet*, making the rule *Net* no longer valid, as depicted in the error tooltip.

3 Formalizing Metamodel/artifacts Co-evolution Relationships

As explained in Section 2, the intent of our approach is to provide an approach methodology that can be useful to predict and make decision on the refactoring impact of the domain metamodel on its related artifacts. The approach can be used as a specification and a guideline for adaptation processes. Below we will explain how the formalization of the relations help to automate and to better document the impact of domain metamodel changes on its related artifacts. The relations between the domain metamodel and its related modeling artifacts are of different nature; however, despite their different role within the modeling artifacts ecosystem, they share a common structure. In this section we present a mechanism to formalize the common structure of those relationships. In the remainder of this section we apply the proposed mechanism to define the *domainConformsTo* relation in Section 3.1, in Section 3.2 we formalize the *dependsOn* relation, and in Section 4 we present the formalization of the *conformsTo* relation.

The common structure of those relations implies that they can be uniformly formalized as a set of links, where each type of link has a different meaning depending on the nature of the

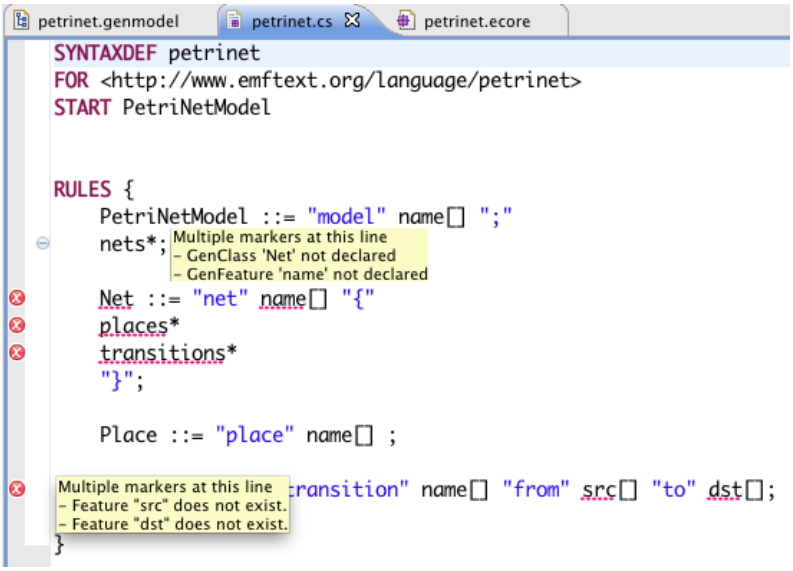


Figure 8 – A EMFText concrete syntax specification impacted by the evolution of the Petri net metamodel

involved relation. Regardless of the adaptation process for managing the modeling artifacts evolution, it is important to precisely define how the various relations among the involved modeling artifacts can be formalized.

In this section we will show that by formalising those relations, we provide support for a new level of automation and for the development of adaptation techniques in a more controlled environment. Indeed, our formalisation allows developers to automatically derive the dependencies between the domain metamodel and its related modeling artifacts, decreasing the risks that may exist when manual checks are performed (like assuming erroneous dependencies between the artifacts, having false positives when considering the impacted modeling elements, and so on).

In Figure 9 we show a general overview of the formalization of the relations between a *generic artifact A* and the *domain metamodel MM*. An *Artifact A* conforms to its *Artifact Metamodel* MM_A, and it has some kind of relation with the *Domain Metamodel MM*; for example a EMFText syntax specification is connected to the domain metamodel by means of its rules and conforms to the EMFText metamodel. Each kind of artifact must conform to a formal specification of the concepts that it can express. An artifact must respect some constraints and can use only concepts expressed in its metamodel: we call that metamodel *Artifact Metamodel* MM_A, where A is the artifact and the relation holding between them is the well-known conformance relation. The relation between the artifact A and the domain metamodel MM, called *Dependency*, can be represented as a set of links. Each link connects elements within the domain metamodel (like metaclasses, properties, etc.) with specific elements in the modeling artifact. These links represent the dependencies which can be used to assess the impact that some change within the domain metamodel may have on elements within the linked modeling artifact. Under this perspective, elements not connected at this stage represent *independence* between the domain metamodel and its related artifact; that is, they represent those elements within the domain metamodel whose evolution will not directly impact any element within the artifact being considered.

In this work the links between any modeling artifact within the software ecosystem are

contained into special models (e.g., $wm_{relation}$ and $wm_{dependency}$ in Figure 9) called *weaving models* [5]. A weaving model can be seen as a means for setting fine-grained relationships between models and executing operations on them[6].

For example, in the ecosystem we may have a weaving model between a model transformation T and its source metamodel MM containing a link between each transformation rule in T and its corresponding input metaclass defined in MM; those links can then be used (i) to establish fine-grained dependency relationships between T and MM, and (ii) to identify which transformation rules need to be reconsidered if some metaclass in MM has been changed.

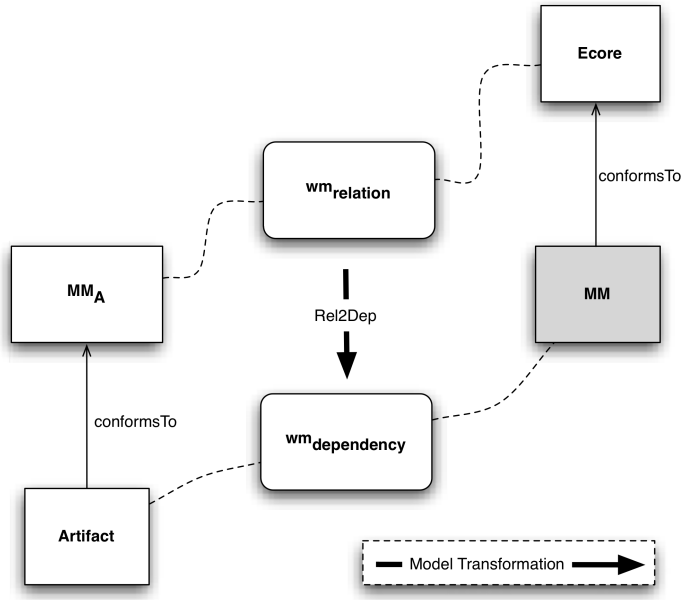


Figure 9 – General overview of the relation definition

In this work, we homogeneously represent the dependency between a domain metamodel *MM* and its related artifacts (*A* in Figure 9) as a weaving model ($wm_{dependency}$ in Figure 9). Representing those links(dependency links) as a weaving model between *A* and *MM* offers a number of advantages; more specifically, it helps in keeping *MM* and *A* loosely coupled, avoiding to have a large metamodel for capturing all the modeling concepts and their relationships. Also, adhering to the "everything is a model" principle [5], those weaving models can be automatically generated by means of model transformation techniques. Indeed, in our approach the links in $wm_{dependency}$ are not defined manually (which may be a very risky and error-prone activity), they are automatically generated. This generation step is automatic only if the relations between the types of the element of *A* and the types of elements of *MM* are known a priori and then formalized. In our approach, we represent this information as an additional weaving model called $wm_{relation}$. Such a weaving model is composed of a set of bidirectional links between the concepts in MM_A and the *types* of the concepts in *MM* which may be linked to *A*, that is the metametamodel that *MM* conforms to. In our approach we use *Ecore* as metametamodel[7]. In doing so, we manage to represent the relations between

[6]For the sake of clarity, we do not show those links in Figure 9, however the interested reader can refer to Sections 3.1, 3.2 and 4.2, in which they will be extensively used and described.

[7]The Eclipse Modeling Framework (EMF) includes a meta model (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.

MM and *A* in a generic way, independently from the various forms that *A* may have. This opens for the possibility to automatically generate $wm_{dependency}$ by means of the generic model transformation *Rel2Dep*. This transformation takes as input $wm_{relation}$, the artifact *A* and the domain metamodel *MM*, and produces as output $wm_{dependency}$ according to the links defined in $wm_{relation}$.

This aspect of our approach is also related to the genericity of the used weaving models. More specifically:

- $wm_{relation}$ is generic since it is defined between the metamodel MM_A specifying the type of a set of artifacts and the Ecore metametamodel. Fundamentally, in the context of a single specific type of artifact (e.g., ATL model transformations), this weaving model will always be the same, independently of how many artifacts of type MM_A exist within the ecosystem.

- $wm_{dependency}$ is defined between each artifact A and the domain metamodel MM. Clearly, this weaving model is not generic since it depends on the specific artifact being considered. In this paper our *Rel2Dep* transformation is needed to alleviate this issue; indeed, *Rel2Dep* allows modellers to automatically obtain $wm_{dependency}$ weaving models by starting from the information defined by the generic $wm_{relation}$ weaving model.

Summarising, the $wm_{dependency}$ weaving model between *A* and *MM* represents the dependencies that identify all meta-elements in *MM* that are connected to elements in *A*; then a refactoring on a meta-element in *MM* may induce a change propagation on the connected element in *A*. In next sections we will show the application of our approach by defining the generic relations for ATL and EMFText.

When considering typical modeling artifacts such as model transformations, concrete syntax specifications, templates for code generation, a recurrent treat is that they are defined in terms of metamodeling elements. For example, input and output patterns of rule in a model transformation are specified as a metaclass with some condition on its elements, a template in a concrete syntax specification specify which pieces of text must be generated for all instances of a given metaclass, etc. This means that for typical modeling artifacts and engines, there is a natural connection between *MM* and their constructs, however this is not the case when we consider the *conformance* relation between the domain metamodel and the terminal models conforming to it. Thus, special attention must be paid to deal with the *conformance* relation; in Section 4 we will describe this specific aspect of our approach.

3.1 Relating ATL Transformations to the Domain Metamodel

As anticipated in Section 2, in this work we call *domainConformsTo* the relation between the domain metamodel and an ATL model transformation. Figure 10 shows how the *domainConformsTo* relation can be formalized by instantiating the generic mechanism described in the previous section.

As represented in Figure 10, the generic artifact *A* has been instantiated with an ATL transformation *T* and the Artifact Metamodel MM_A has been instantiated with the ATL metamodel. The weaving model $WM_{relation}$ between the ATL metamodel and Ecore specifies which meta-elements in ATL are connected to other meta-elements in Ecore; for example the *OclModelElement* concept in ATL is connected with the *EClass* concept in Ecore through the *name* of both of them, and so on. At this point, the $wm_{dependency}$ weaving model containing the dependencies between the transformation *T* and the domain metamodel *MM* can

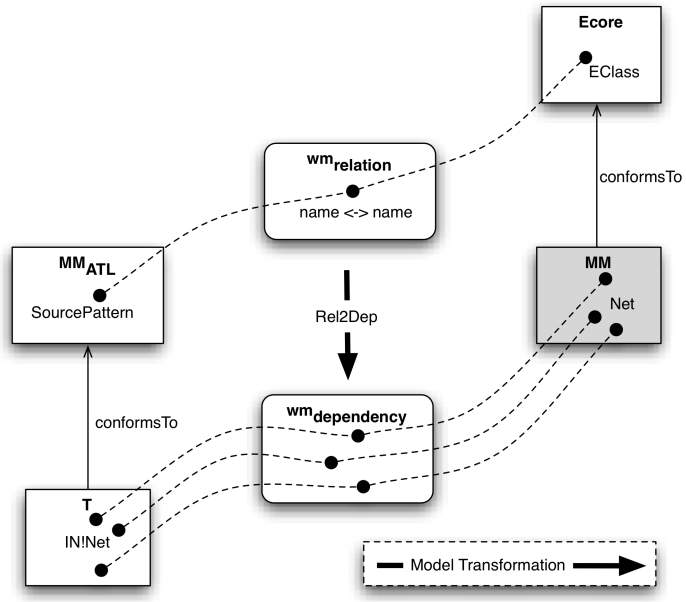


Figure 10 – Relating ATL transformations to the domain metamodel

be automatically generated. This is done by executing our generic *Rel2Dep* model transformation that takes as input $wm_{relation}$, *T*, and *MM* and produces the $wm_{dependency}$ weaving model containing the dependencies between *T* and *MM*. For example, the *OclModelElement* with the property name "Net" is connected to the metaclass (which is an instance of *EClass* in Ecore) in the PetriNet metamodel with the name "Net". This dependency link specifies that changing the name of the *Net* metaclass in the PetriNet metamodel implies to propagate such a change to each *OclModelElement* linked to it.

3.2 Relating Concrete Syntax Specifications to the Domain Metamodel

Similarly to what has been done for ATL transformations, in this section we show how our approach can be used for defining the relation between EMFText and the Ecore metamodel by creating another $WM_{relation}$ weaving model. Again, the $WM_{relation}$ weaving model, together with a EMFText syntax specification and the domain metamodel *MM* will be the input of the *Rel2Dep* transformation. The result of this transformation is the $WM_{dependency}$ weaving model linking the specific EMFText syntax specification to the domain metamodel *MM*. Figure 11 shows that the generic artifact *A* has been instantiated with a EMFText syntax specification *S* for the Petri net metamodel and the Artifact Metamodel MM_A has been instantiated with a metamodel containing the definition of the constructs of EMFText.

In this case, the links between the domain metamodel and the EMFText meta-elements are done by name (e.g.Package, Rule, children, are meta-elements of the EMFText metamodel connected to the domain metamodel), therefore the relationship between them can be formalized by considering pairs of matching names. For example we identified a link between the *EClass* concept in Ecore and the concept of *Rule* in EMFText by passing through their *metaclass* property. Similarly to what happened for ATL transformations, changing the

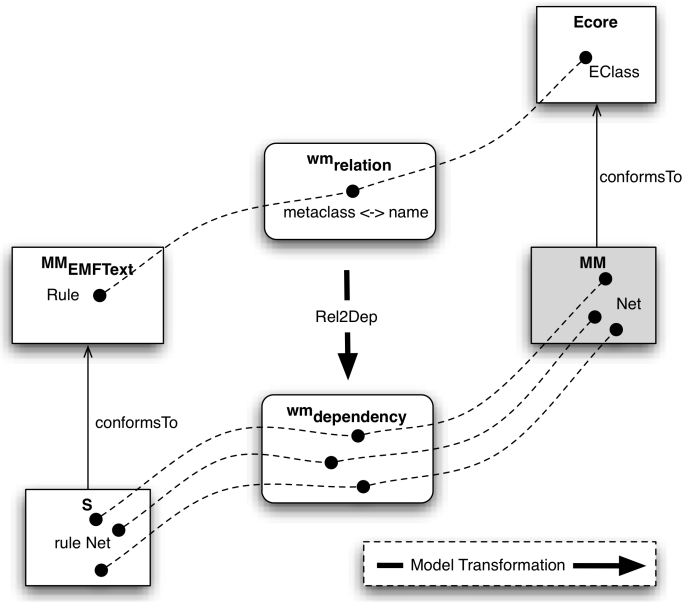


Figure 11 – Relating EMFText specifications to the domain metamodel

name of a metaclass in *MM* induces the corruption of the syntax specification; in particular, each *Rule* linked to the changed *EClass* must be changed.

4 Managing the Conformance

As anticipated in Section 3, the *conformsTo* relation must be considered differently from the other relations. Indeed, conformance is more strict because, differently from the other relations for typical modeling artifacts in which there is a natural connection between the domain metamodel *MM* and their constructs, it is a cross-level relation in the metamodeling stack. In addition, if we consider Figure 12, each considered artifact (i.e., *T*, *S*, *A* in the figure) depends on the domain metamodel *MM*; that is, if the domain metamodel *MM* undergoes some change, those artifacts may be impacted by the changes and may need to be adapted accordingly. Also, it should be noted that each modeling artifact conforms to its corresponding metamodel (i.e., MM_{ATL}, $MM_{EMFText}$, and MM_A in the figure), which we assume it does not change during the development process. On the contrary, if we consider model instances (i.e., *m* in the figure), the dependency relationship between them and the domain metamodel is the *conformsTo* relation itself. Naturally, each model instance *m* conforms to the domain metamodel itself, which actually is the subject of the evolution (that is, it can change during the development process). So, the main difference between the *conformsTo* relation and all the other relations is that when we consider any modeling artifact in the ecosystem, the metamodel of the element that is impacted by a change in *MM* is fixed, whereas when we consider instances of the domain metamodel, their metamodel is exactly the element that has changed.

Conformance is implemented in the EMF[8] architecture and is analogous to the "Class/in-

[8]http://www.eclipse.org/modeling/emf/

stance" relationship in Java programs.

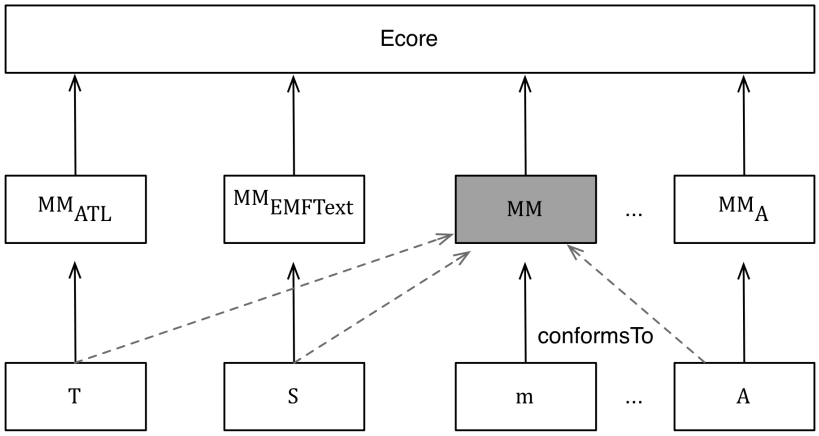


Figure 12 – Relation between Ecore and tool metamodels

As represented in Figure 12, the *conformsTo* relation we are considering in this part of our work is the one between the domain metamodel *MM* and the terminal models conforming to it (*m* in figure). When linking the other artifacts to the domain metamodel (e.g., the transformation T with *MM* in figure), we manage to define their relationship as a weaving model between the metamodel of the involved artifact (MM_{atl} in figure) and the Ecore metametamodel. However, we cannot directly apply this mechanism when considering the relation between *MM* and the terminal models conforming to it. This is because the domain metamodel is (i) subject to evolution and (ii) variable for each domain being considered; thus, we cannot assume that the domain metamodel *MM* is fixed. Basically, a possible solution for this issue is to define a mechanism for representing terminal models in a metamodel-independent manner. In next sections we show how to represent model instances in a metamodel-independent representation, and then we present how we exploit this mechanism for explicitly defining the *conformsTo* relation.

4.1 A metamodel-independent Models Representation

In order to represent model instances independently from the metamodel they conform to, we need a generic metamodel for specifying model instances; we call it *GM*. Even though for different purposes, this technique has been used also in other approaches in MDE research. For example, in [35] a domain-specific (meta)metamodel has been introduced as well; however, the metamodel presented in [35] contains concepts for describing both models and metamodels, whereas our *GM* metamodel contains only concepts for describing model instances. This is a consequence of the different purposes of the two approaches: the metamodel presented in [35] is used for simplifying the representation of model differences in a generic manner, whereas our *GM* metamodel has been designed to have a generic representation of a model, independently from the metamodel it conforms to. Also, our approach is in line with the approach defined in [29], where a metamodel independent syntax for models has been defined for automatic consistency checking. It is important to note that our *GM* metamodel is different from a metametamodel because:

- a metametamodel is a model that is its own reference model (i.e., it conforms to itself) [21]; instead, our *GM* metamodel conforms to the Ecore metametamodel only

(which is different from the *GM* metamodel itself);

- our *GM* metamodel exists at the M2 level of the metamodeling stack, whereas metametamodels exist at the M3 level. More specifically, metametamodels play the role of reference models for metamodels, whereas our *GM* metamodel is the reference model for terminal models only.

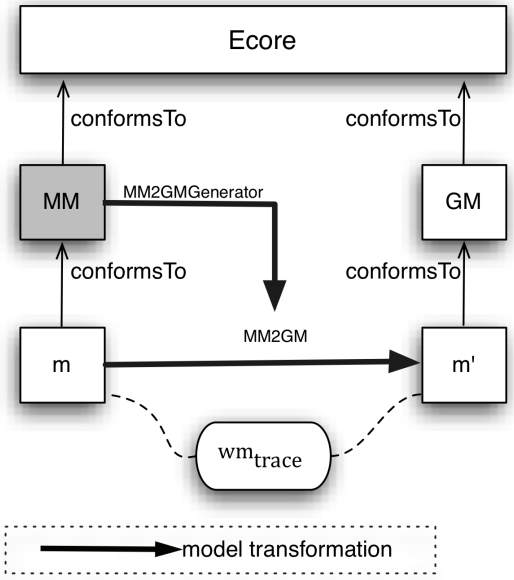


Figure 13 – General models representation

Figure 13 shows how an instance of the *GM* metamodel (*m'* in the figure) relates to its corresponding instances of the domain metamodel *MM* (*m* in the figure). Fundamentally, *m'* contains the same entities of the *m* model; they differ in the metamodel they conform to: *m* conforms to the domain metamodel (which may be different for each domain under consideration), whereas *m'* conforms to the generic *GM* metamodel (which is unique, regardless of the domain under consideration).

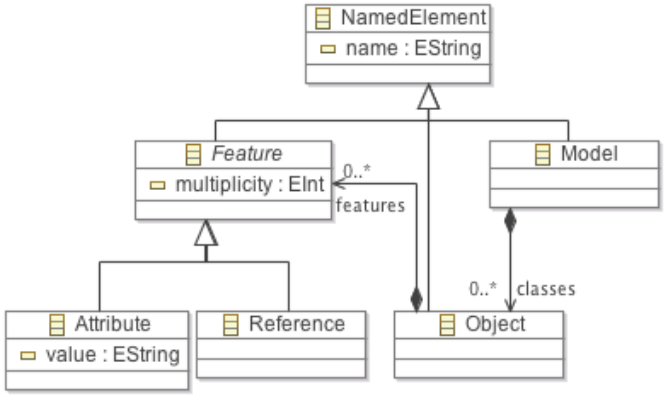


Figure 14 – Generic metamodel for model instances

Since *GM* is unique, we can provide a generic higher-order transformation[9] *MM2GM Generator* that takes as input a domain metamodel MM and produces a model-to-model transformation *MM2GM*. The *MM2GM* transformation allows developers to automatically obtain a generic model instance *m'* starting from a standard model *m*. The *MM2GM* transformation creates also an auxiliary weaving model containing tracing information between model elements in *m* and the obtained model elements in *m'*; this tracing information will be useful while performing change impact analysis because they allow us to identify which elements in *m* correspond to specific elements in *m'*.

It is important to note that the *MM2GM* transformation is not generic since it actually depends on the domain metamodel *MM*. Indeed, if we consider *MM2GM* as a function, it can be defined as follows:

$$MM2GM\colon MM \rightarrow GM\ x\ MM_{trace}$$

where MM is the domain metamodel, GM is our generic metamodel for model instances, and MM_{trace} is the metamodel of the tracing weaving model. Clearly, $MM2GM$ depends on the specific domain metamodel MM under consideration. It is also important to note that the higher-order transformation *MM2GMGenerator* is generic, and always applicable independently from the considered domain metamodel *MM*. This allows us to reach genericity by automatically generating the $MM2GM$ transformation by means of the $MM2GMGenerator$ higher-order transformation. Intuitively, if we consider $MM2GMGenerator$ as a function, it can be defined as follows:

$$MM2GMGenerator\colon Ecore \rightarrow MM_T$$

where $Ecore$ is the unique metametamodel and MM_T is the metamodel of a model transformation language (e.g., the metamodel of ATL). In this case, there is no dependency between $MM2GMGenerator$ and either the domain metamodel MM or any model conforming to it.

Figure 14 shows a fragment of *GM*. It is composed of a root *Model* concept, that contains a set of *Object*s representing instances of some metaclass defined in a metamodel; each *Object* can contain zero or more *Feature*s. A *Feature* can be either an *Attribute* or a *Reference*, depending on whether the type of the feature is primitive or an object. More specifically, strings, integers or booleans are primitive types, whereas objects are instances of some metaclass of the metamodel. All the elements within a model conforming to *GM* can have an associated identifier, it is defined by means of the *name* property in *NamedElement*. As an example of a possible instance of *GM*, in Figure 15 we show the metamodel-independent version of the example model represented in Figure 6. This model contains the same information of the Petri net model in Figure 6: a *net1* object containing four objects *p1,p2,p3,t1* in relation between them. For the sake of brevity, we do not go into further details about the shown Petri net model.

It is important to note that generic model instances defined by using *GM* are not directly linked to their corresponding metamodel (that in our case is the domain metamodel *MM*). This is exactly what we want to achieve in this context: model and metamodels are not directly coupled and they do not depend on each other. This allows us to treat model instances

[9] An higher-order transformation is a special kind of model transformation taking other transformations as input or producing other transformations as output.

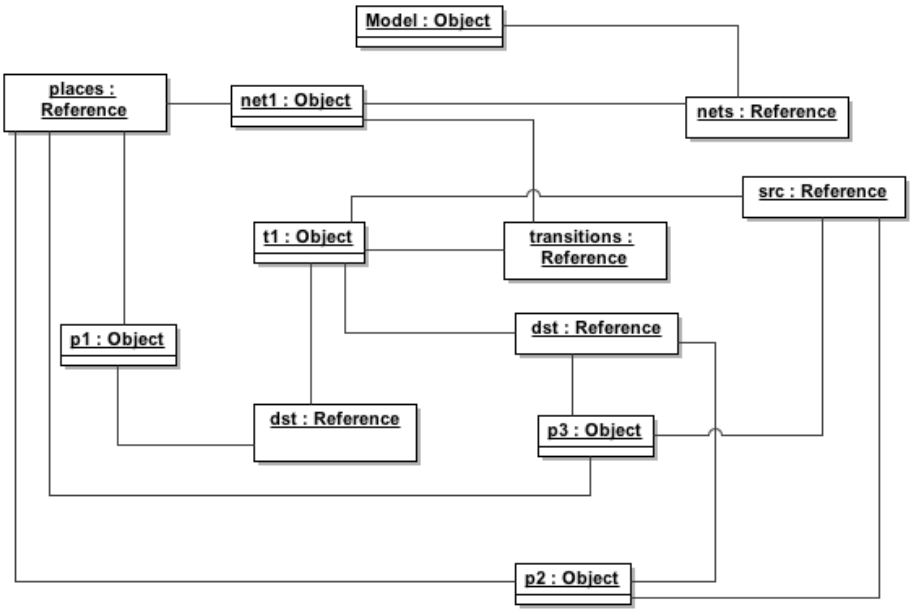


Figure 15 – PetriNet model representation with GM

in the same way as we treat all the other modeling artifacts depending on the domain metamodel.

4.2 Relating Models to their Metamodel

By using the generic metamodel described in Section 4.1, we can explicitly define the conformance relation between a model instance in *GM* and the domain metamodel *MM* through our $wm_{relation}$ and $wm_{dependency}$ weaving models (see Section 3). This allows us to treat instances of the domain metamodel *MM* in the same way as we treat all the other modeling artifacts. More specifically, in Figure 9 we represented the overview of how we treat the relation between a generic modeling artifact with respect to the domain metamodel *MM*; in this context *m* is a model instance, generally represented in XMI (i.e., the standard XML Metadata Interchange). A model *m* conforms to the domain metamodel *MM*.

Figure 16 shows how we represent the "conformsTo" relation by means of the weaving models described in Section 3. The $wm_{relation}$ weaving model is defined between the Ecore metamodel and the generic *GM* metamodel. The weaving links contained in $wm_{relation}$ formalize the conformance relationship independently from the domain metamodel MM. Figure 17 shows how we defined such a relationship in $wm_{relation}$. In this case, the weaving links in $wm_{relation}$ represent a refinement of the "conformsTo" relation between Ecore and GM. For example, a weaving link between *EClass* in Ecore and *Object* in *GM* establish that each object in an instance of *GM* is an instance of some metaclass in a metamodel conforming to Ecore. Similarly, weaving links exist also between *EReference* in Ecore and *Reference* in *GM* and between *EAttribute* in Ecore and *Attribute* in *GM*.

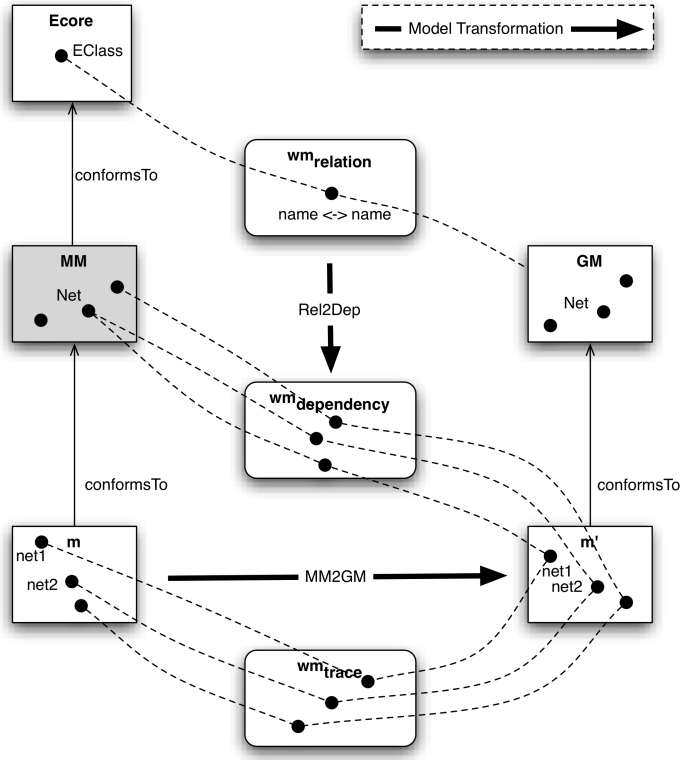


Figure 16 – General overview of the relation definition

In the same way as ATL and EMFText are managed in Section 3, the $WM_{relation}$ weaving model, together with an instance of *GM* and the domain metamodel *MM* will be the input of the *Rel2Dep* transformation. The result of this transformation is the $WM_{dependency}$ weaving model linking the specific model instance of *GM* to the domain metamodel *MM*. In this context, $WM_{dependency}$ contains the information on impact analysis; for example, $WM_{dependency}$ specifies that a change in the MetaClass "Net" in *MM* will impact each Object of type "Net" in *m'*. The tracing information stored in the previously generated wm_{trace} weaving model (see Section 4.1) will identify all the affected elements in the original model *m* conforming to the domain metamodel *MM*.

5 Megamodeling for Managing Co-evolution

Megamodeling is a promising technique for representing and managing relationships that may exist between a set of modeling artifacts. It has been introduced for the first time in 2004 and it actually opens to interesting new possibilities for exploiting standard model-driven techniques also in complex situations in which large sets of interrelated models must co-exist. In the remainder of this section, Section 5.1 provides a general overview on what megamodeling is, Section 5.2 motivates why megamodeling plays a fundamental role in our approach and the added values it provides, and Section 5.3 describes how we manage the co-evolution of interrelated modeling artifacts by means of megamodeling techniques.

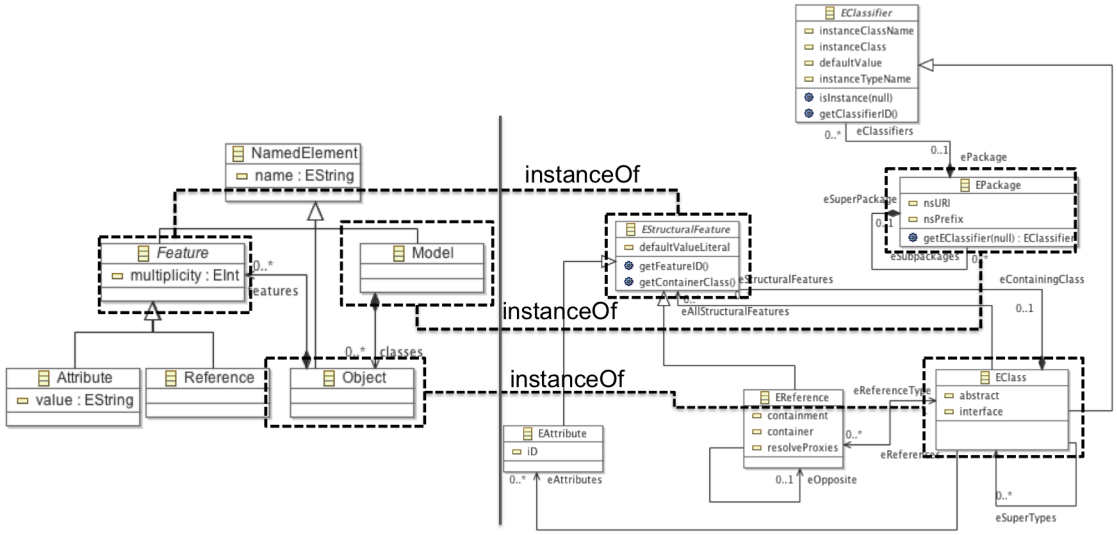


Figure 17 – ConformsTo relation weaving model

5.1 What is a Megamodel?

Megamodeling [7] has been proposed with the aim of supporting modeling in the large, i.e. dealing with models, metamodels, and their properties and relations. Intuitively, a megamodel is a model of which at least some elements represent and/or refer to models or metamodels. While a metamodel specifies properties and rules governing models construction, a megamodel specifies properties and rules governing Model Driven Engineering (MDE) artifacts construction, and among them, models and metamodels. Megamodeling operations support the management of large libraries of artifacts that, as introduced before, could also be models and metamodels. Megamodeling offers the possibility to specify relationships between models (and metamodels) and to navigate among them.

As a concrete implementation of megamodeling, we refer to the AMMA platform[10], as presented in [6]. More specifically, a dedicated component of the AMMA platform is in charge of managing megamodels, it is called AM3 [1].

Figure 18 shows a fragment of the AM3 megamodeling conceptual framework, as presented in [1]. In the following of this paper we refer to a metamodel of a megamodel as *metamegamodel*. The main types of models are: (i) *Terminal Model*s, which represent the real system and conform to some metamodels, (ii) *MetaModel*s, which define domain-specific concepts and conform to metametamodels, (iii) *MetaMetamodel*s, which provide generic concepts for metamodels specification and conform to themselves.

Figure 18 shows different kinds of terminal models: (i) *Transformation Model*s, which are used to define transformations between models, (ii) *Weaving Model*s, which are used to define relationships among models, and (iii) *MegaModel*s, which are used to support the megamodeling process.

Megamodeling allows also to specify (typed) relationships between the modeling artifacts contained into a megamodel. Figure 19 shows another fragment of the megamodeling conceptual framework representing the artifacts relationships. They represent the generic re-

[10] http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT

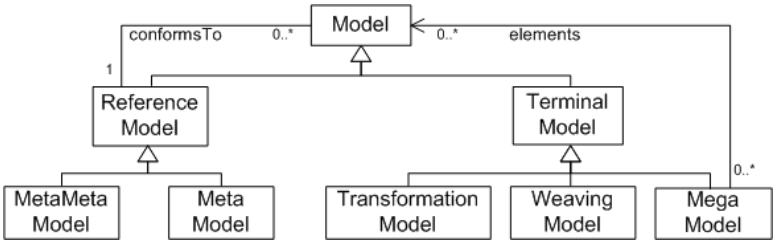


Figure 18 – AM3 Megamodeling Conceptual Framework: basic elements

lationships that may exist between the various entities within the megamodel. An *Entity* is the root metaclass of the generic metamegamodel, and represents any possible concept that may exist within a megamodel. A relationship may be either bidirectional (*Relationship* in Figure 19) or directed (*DirectedRelationship* in Figure 19), i.e., if it distinguishes or not between the elements it links.

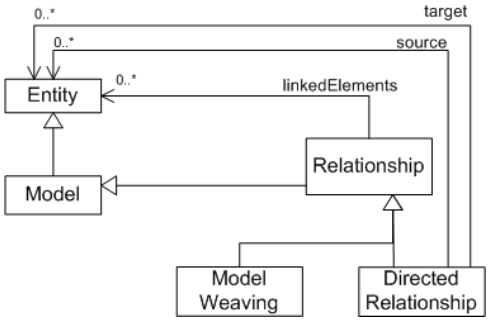


Figure 19 – AM3 Megamodeling Conceptual Framework: relationships

In many cases it is important to express relationships between modeling artifacts at two different levels of abstraction: between models and between model elements within the models. To support these two levels, both model-level relationships and model element-level relationships must be defined. Thus, AM3 introduces a special kind of relationship to represent model-level relationships, it is called *ModelWeaving* and it is refined into a weaving model (*WeavingModel* in Figure 20) that represents model element-level relationships. Please refer to [22] for more information on this aspect of the AM3 megamodeling conceptual framework.

In AM3, a megamodel records all available resources and acts as an MDE repository. From a practical point of view, AM3 manages megamodel elements (for example model transformations, tools, UML models, metamodels) and provides a user interface to manipulate them. This user interface is generic since it does not depend either on the models, metamodels or other artifacts within the megamodel. AM3 is extensible, and thus it allows developers to extend the base metamegamodel by providing new concepts in a separate metamegamodel.

Under this perspective, co-evolution related concepts can be defined as an extension of the base metamegamodel; in doing so, it is possible to navigate modeling artifacts and establish (typed) relationships among them by reusing the AM3 extensible platform. This is essentially what we will present in Section 5.3.

5.2 Why Megamodeling for Managing Co-evolution?

Megamodeling offers the possibility to handle models (and metamodels) as first-class entities, to specify relationships between them and to navigate among them. This is fundamental in our approach since all the involved modeling artifacts have different relationships defined among them and towards the domain metamodel.

Firstly, it is important to note that a megamodel is a model itself, and thus it conforms to a specific metamodel (our metamegamodel for managing the co-evolution of modeling artifacts is presented in next section). This aspect of megamodeling is in line with the classical *everything-is-a-model* MDE principle and it opens for new interesting possibilities for managing the co-evolution of modeling artifacts. Intuitively, a megamodel can be seen as a model representing a complete map of the ecosystem of modeling artifacts under consideration. Thus, it can be manipulated as a standard model: defining OCL rules, predicating on it and checking specific properties (as we do for example in Section 5.4), defining model transformations taking it as input and producing some other model, or defining model transformations which may refactor the megamodel in some way, etc.

Furthermore, by keeping track of all the heterogeneous modeling artifacts within a megamodel, we treat all of them as models (each of them conforming to their corresponding metamodel). This results in an homogeneous infrastructure which enables the management of complex artifacts; indeed, that complexity is defined and encoded into their metamodels, thus enabling programmatic management of (even complex) models.

As introduced in the previous section, megamodeling offers the possibility to specify relationships between models (and metamodels) and to navigate among them. This is fundamental for the co-evolution approach we are proposing since it allows us (i) to seamlessly identify the possible dependencies that exist between modeling artifacts, and (ii) to consider those dependencies in order to assess the impact of a change triggered in a specific artifact. Moreover, in our approach we define those relationships as weaving models (i.e., $wm_{relation}$ and $wm_{dependency}$ in Section 3). As a consequence, in our approach we can basically define relationships between modeling artifacts and the domain metamodel at two different levels on granularity: on one hand the megamodel represents all the modeling artifacts involved in a given context as well as their various relationships with the domain metamodel; on the other hand, weaving models are used to represent finer-grained relationships between the various elements contained in the involved models. Summarizing, using megamodels and weaving models in combination gives a good level of flexibility to our approach. Indeed the megamodel allows modellers to check which modeling artifacts depend on the domain metamodel, where weaving models allow to "zoom" into the considered models and to check which elements within the models are dependent to which elements within the domain metamodel.

5.3 Megamodeling for Managing Co-evolution

In this section we provide a new extension to the AM3 base megamodel that is specifically focused on the management of the co-evolution of modeling artifacts. By combining model weaving and megamodeling, we aim to provide a generic infrastructure for managing the co-evolution of modeling artifacts and for developing new adaptation techniques that can build on our infrastructure.

A fundamental step for building our generic approach is the creation of a generic metamodel encompassing the concepts for representing the various relationships defined in Sections 3 and 4; we call this metamodel GMM4EVO, it stands for Global Model Management for managing co-EVOlution. Megamodels conforming to GMM4EVO have other models as

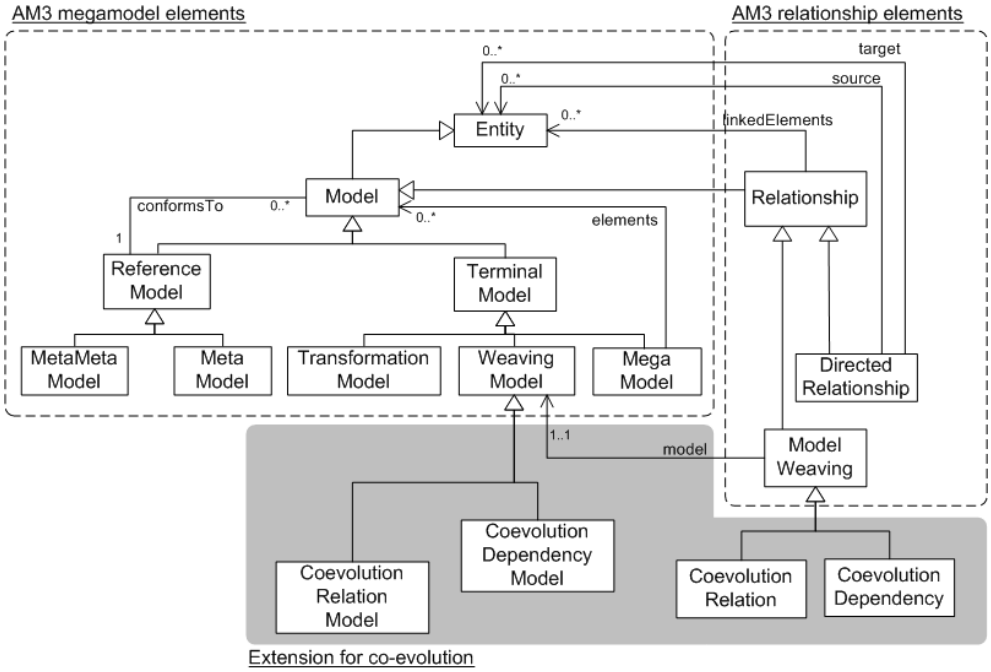


Figure 20 – Extract of the metamegamodel extension for co-evolution

first class entities, such as terminal models, their metamodels, models for representing model transformations, graphical representations, and so on; the weaving models and weaving metamodels defined in the previous sections of this paper are also part of the megamodel.

Figure 20 presents an excerpt of GMM4EVO. GMM4EVO is defined as an extension of the generic meta-megamodel provided by AM3. The resulting metamodel can be considered as composed of three main parts:

1. **AM3 megamodel elements**: they represent all the elementary elements in MDE, that is, terminal models, metamodels, model transformations, weaving models, etc. They have already been described in Section 5.1.

2. **AM3 relationship elements**: they represent the generic relationships that may exist between the various modeling artifacts within the megamodel. They have already been described in Section 5.1.

3. **Extension for co-evolution**: this part of the metamegamodel represents our extension for managing the co-evolution of modeling artifacts in a generic manner. According to what said in the previous point, we define relationships at both the modeling level and the model-elements level. More specifically, our extension is composed of four metaclasses:

   - **CoevolutionRelation**: specialization of the *ModelWeaving* metaclass that is used to link the metamodel of a specific kind of artifact to a metametamodel.
   - **CoevolutionRelationModel**: it is an extension of *WeavingModel* and represents a refinement of a coevolution relationship. Within the megamodel, instances of this metaclass are weaving models that define correspondences between elements of

the metamodel of a specific kind of artifact (e.g., the metamodel of ATL) and the elements of a metametamodel. As described in Section 3, these correspondences can be used to automatically generate the correspondences between instances of the linked metamodel (e.g., a specific model transformation written in ATL) and the domain metamodel itself.

- **CoevolutionDependency**: it is a specialization of the *ModelWeaving* metaclass and it represents a link at the modeling level between the metamodel of an involved artifact and the domain metamodel.
- **CoevolutionDependencyModel**: this concept extends *WeavingModel* and it is a refinement of a *CoevolutionDependency*. An instance of this metaclass is a weaving model containing the various correspondences between an involved modeling artifact (e.g., a model transformation) and the domain metamodel. Those weaving models can be considered as the building blocks for assessing the impact that a change in the domain metamodel can have on the other woven metamodels.

Developing a dedicated metamegamodel for managing the co-evolution of modeling artifacts allows us to (i) represent modeling artifacts at all levels of abstraction (i.e., modeling, metamodeling, and metametamodeling levels) in a homogeneous manner, (ii) define the various relationships and correspondences between all the involved modeling artifacts in a generic and natural way, and (iii) build an infrastructure for developing future engines that may manage and navigate the megamodel firstly for assessing the impact of a change within the involved modeling artifacts and secondly for (automatically) adapting the involved modeling artifacts in response to the occurred changes.

Section 6 discusses a complete scenario in which a megamodel conforming to GMM4EVO is defined for managing the co-evolution of the domain metamodel and an ATL model transformation.

5.4 Change Impact Analysis

In their seminal book, Bohner and Arnold defined change impact analysis as *"identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change"* [2]. In the context of our work, change impact analysis can be considered as the activity of detecting which modeling artifacts within a megamodel conforming to GMM4EVO are impacted by a change made in the central domain metamodel.

Since our megamodel provides a detailed map of the relations between all the involved modeling artifacts, identifying all the modeling elements impacted by a change in the domain metamodel is a straightforward activity. Listing 2 presents an OCL constraint that runs on the megamodel and identifies all the impacted modeling elements. The main part of the listing is the *impactedElements* ATL helper (line 1 in the listing). It is defined for a `Metamodel` in the megamodel which represents the domain metamodel *MM*, and takes as input a set of modeling elements (`changedElements` in the listing) which have been previously changed during the metamodel evolution. The output of the *impactedElements* helper is the set of model elements which may be impacted by the changes made in the domain metamodel. From an high-level point of view, the logic of the *impactedElements* helper is to return all the model elements in the megamodel that are connected by means of some dependency weaving link to at least one element in the `changedElements` set.

Listing 2 – OCL rule for change impact detection

```
1 helper context AM3!Metamodel def : impactedElements(changedElements : Sequence(ECORE!
      EObject)) : Set(ECORE!EObject) =
```

```
2 self.getLinkedCoevolutionModels()
3 ->collect(e | e.links->select(e2 | changedElements.contains(e2.source))
4  ->collect(e3 | e3.target.getReferredElement()))
5  .flatten().asSet();
6
7helper context AM3!Metamodel def : getLinkedCoevolutionModels() : Sequence(AM3!
      CoevolutionDependencyModel) =
8 AM3!CoevolutionDependency.allInstancesFrom('MEGAMODEL')->select(e | e.linkedElements.
      contains(self))
9  ->collect(e | e.model);
```

In line 2 of Listing 2 the megamodel is navigated in order to get all dependency weaving models which are connected to the domain metamodel (i.e., *self* in the listing); this is done by means of the auxiliary helper called *getLinkedCoevolutionModels()*. Then, in line 3 we collect all the weaving links within the identified weaving models which have an element in *changedElements* as source; at this point, in line 4 we consider all the target elements of those weaving links: they represent the model elements impacted by the changes in the domain metamodel. The *flatten()* and *asSet()* operations in line 5 of the listing are used to create a single sequence starting from the nested sequences obtained so far, and to merge possible duplicated elements within the flattened sequence, respectively.

The OCL constraint described in Listing 2 is implemented as an ATL helper since the the proposed approach is based on the AMMA platform (ATL is part of it as well) and ATL provides a stable and intuitive implementation of the OCL language; further on, ATL queries can be launched either programmatically or via Ant scripts[11]. This results in a homogeneous framework to orchestrate, manage and configure such queries within the megamodel. The result of this kind of queries can be either stored in an external file, serialized into a specific model or simply printed to the Eclipse console.

Obviously, change impact detection does not come alone: once the impacted elements have been identified, a series of adaptation processes must be performed in order to re-establish a certain consistency of all the modeling artifacts with respect to the evolved domain metamodel. Under this perspective, the focus of this work is on the detection of the elements that may be impacted by the evolution of the domain metamodel; the proposed approach is complementary to the used adaptation techniques, and thus the study of the adaptation techniques that may be applied are out of the scope of this work.

6 Example

In this section we describe an example in which our approach is applied on the Petri net scenario described throughout the paper. For the sake of clarity, in this example we focus exclusively on the management of a specific modeling artifact: the Petrinet2PNML model transformation. Being generic, our approach can be applied to manage other kinds of modeling artifacts in the same way as we will show in the remainder of this section.

The process we will follow in this example scenario can be summarized as follows:

1. populate the megamodel with the Petri net domain metamodel and the ATL metamodel;
2. define the $wm_{relation}$ weaving model linking the Ecore meamodel with the ATL metamodel;
3. include the Petrinet2PNML ATL transformation;

[11] http://ant.apache.org

4. automatically generate the $wm_{dependency}$ weaving model by means of the generic Rel2Dep higher-order transformation; it will contain tracing information between the Petri net domain metamodel and its dependent Petrinet2PNML transformation.

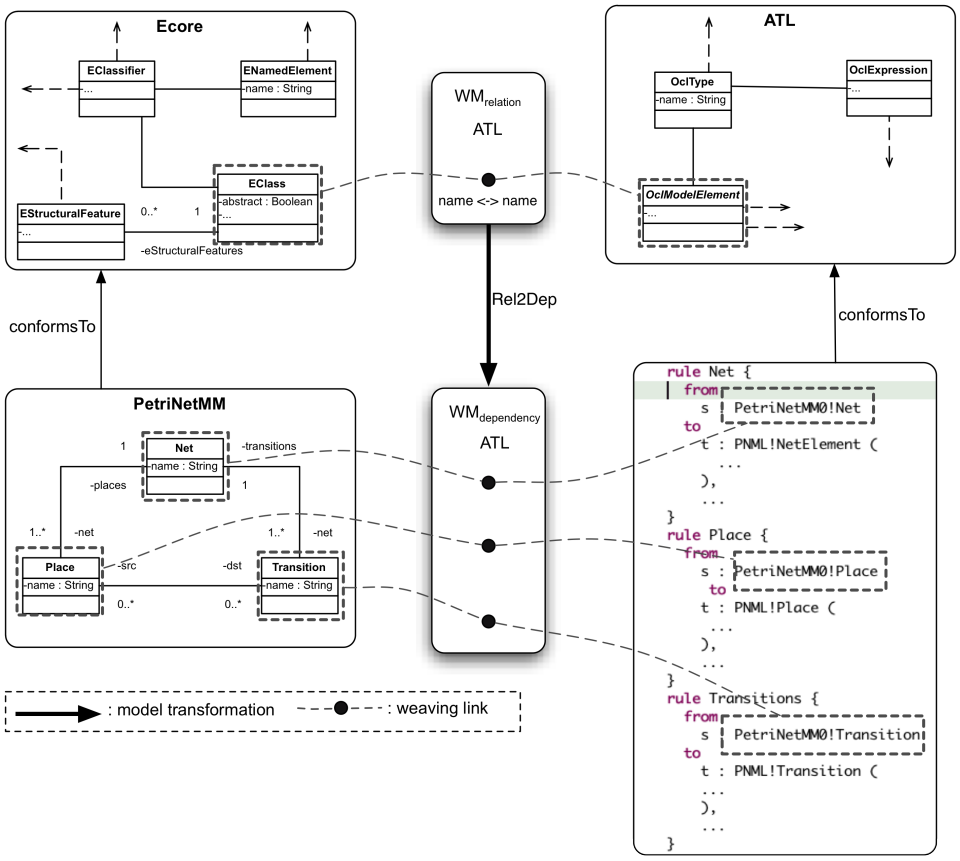


Figure 21 – Example of scenario and weaving models to represent relations

In the remainder of this section we will describe more in detail each step of the above listed scenario and we will apply the mechanism described in Section 3.1. Figure 21 shows the various modeling artifacts involved in this scenario. More specifically, at the beginning the megamodel is populated with the Petri net domain metamodel and the ATL metamodel, together with other auxiliary modeling artifacts like the Ecore metametamodel, etc. The upper-left part of the figure contains a fragment of the *Ecore* metametamodel (for the sake of simplicity, we show only the metaclasses that are relevant for our example). The *PetriNetMM* metamodel contains a simplified version of all the concepts related to a Petri net; according to this metamodel, a Petri net is composed of a net containing both places and transitions which connect places. The *PetriNetMM* metamodel conforms to the *Ecore* metametamodel.

As next step, we link together the *Ecore* metametamodel and the *ATL* metamodel (i.e., a metamodel representing all the concepts that can make up an ATL model transformation); in particular, the link is realized as a weaving model ($wm_{relationATL}$ in the figure) containing a set of weaving links that specify which elements in the *Ecore* metametamodel correspond to other elements in the *ATL* metamodel. For example, if a metaclass (that is, an *EClass*) in the *Ecore* metamodel is changed, this change must be reflected on the *OclModelElement*[12]

[12]OclModelElement is a meta-element of the ATL metamodel, that identify the terminal element of a matching

meta-element with the same name of the *EClass*. In our megamodel, $wm_{relationATL}$ is an instance of the *CoevolutionRelationshipModel* metaclass described in Section 5.3.

Once the $wm_{relationATL}$ weaving model has been defined, we can consider include the *Petrinet2PNML* transformation into the megamodel. The *Petrinet2PNML* transformation is taken from the ATL Zoo[13] and it is designed to generate PNML[14] models starting from Petri net models.

At this point, the generic *Rel2Dep* model transformation of our approach can be executed by taking as input the $wm_{relationATL}$ weaving model, the *PetriNetMM* metamodel, and the *PetriNet2PNML* transformation. The output of this transformation is the $wm_{dependencyATL}$ weaving model which contains the dependencies between the *PetriNetMM* metamodel, and the *PetriNet2PNML* transformation. The $wm_{dependencyATL}$ weaving model allows modellers to keep track of the various dependencies that exist between the *Petrinet2PNML* transformation and the *PetriNetMM* metamodel, which is in this case the domain of the model transformation. In our megamodel, $wm_{dependencyATL}$ is an instance of the *CoevolutionDependencyModel* metaclass described in Section 5.3.

By considering the $wm_{dependencyATL}$ weaving model, we can navigate its weaving links in order to assess the impact that a change in the *PetrinetMM* metamodel can have on the *Petrinet2PNML* transformation. Going down to the lower part of the figure, we can see that the $wm_{dependencyATL}$ weaving model contains three different weaving links. More in details, the *Net* metaclass in the *PetriNetMM* metamodel is connected to the *OclModelElement* with the name *Net* in the *Petrinet2PNML* transformation; the same reasoning applies to the dependencies between the *Place* and *Transition* metaclasses and the *OclModelElement*s with the name *Place* and *Transition*, respectively.

In conclusion, in this section we explained a basic usage scenario of the proposed approach with the main aim of showing the mechanisms underlying our approach from a practical point of view. In this section we did not focus on the graphical user interface (i.e., the GUI) exposed by our approach since currently we are reusing the default GUI of the technologies we use, such as AM3, AMW, ATL and EMF. The interested reader can refer to [3], [11] and [20] to check out some screenshots of their related tools; improving the GUI of our approach is out of the scope of this work, we are evaluating to work on this direction as future improvement of the proposed approach.

It is important to note that these mechanisms are generic since they do not depend neither on the *PetriNetMM* domain metamodel nor on the involved artifacts that depend on it (i.e., the *Petrinet2PNML* transformation). Furthermore, these mechanisms can be applied many times in more complex setups, in which the co-evolution of different kinds of modeling artifacts can be managed by simply navigating the various weaving links that exist between the involved modeling artifacts within the megamodel. In this context, the objectives of having a megamodel are that of (i) keeping organized the various modeling artifacts, (ii) supporting the definition and navigation of the relationships between all the modeling artifacts, and (iii) providing an infrastructure for managing the co-evolution of the domain metamodel and its dependent modeling artifacts.

7 Related Work

The techniques proposed in this paper are related to the research on co-evolution in model-driven engineering [17, 27]. Much of this work is concerned with co-transforming artifacts in

rule(in input or output), e.g. PetriNet!Net

[13] `www.eclipse.org/m2m/atl/atlTransformations/`

[14] PetriNet Markup Language

reply to metamodel changes [36, 28, 19, 9, 32, 28, 24, 13]. Also [12] discusses about metamodel refactorings and artifacts co-evolution introducing ingredients to deal with migration. All of this approaches are based on single artifact type and they propose approach to adapt without alert the developer after metamodel changes. Some changes may be avoided if the impact on the artifacts is too heavy or may create unexpected inconsistencies. For this reason we think that can be very powerful to estimate the impact on artifact before to apply any approach to adapt it.

Using these techniques applied to models/metamodel co-evolution, we need a metamodel-independent representation for models; in this context we use a similar approach to the one defined in [29]. Our approach differs in some details; for example the author defines the *Slots*, that are used to represent values in the model, and the feature attribute indicates the metafeature that the Slot intends to instantiate. Also in [16] the authors propose an independent syntax to manage models, but for a different purpose (i.e., the automatic translation of EMF models into ASP code).

The authors in [30] propose a change impact analysis for Object-Oriented programs. This work provides feedback on the semantic impact of a set of program changes. This analysis is used to determine the existing test programs affected by a set of changes. Using the example of Java classes and existing test cases, adding a method to an existing class may affect the virtual method calls throughout an existing test case. The authors identified also a catalogue of atomic changes. Identifying the changes in Java classes that may be responsible for test failure, these changes could be incorporated safely. There is a similarity between Java classes/test cases and metamodel/artifacts. In our approach the relation between artifact and domain metamodel has to be generically defined in order to identify the dependencies existing between them, whereas in [30] the relation is exclusively tailored to the Java language.

The authors in [34] propose a very promising approach for traceability visualization of model transformations. The proposed tool is called *TraceVis* and it provides a very powerful visualization mechanism for connected models. The authors propose the impact analysis as a possible future application of TraceVis, and in particular of the impact of changes in the source metamodel of a model transformation. This work confirms the important open issue in change impact analysis in the context of metamodel evolution. There is also some related work using megamodeling techniques, testifying that megamodeling can be successfully used in contexts in which models need to be navigated, composed, managed and represented in different ways. The work presented in [18] applies megamodeling techniques to the model driven performance engineering process. In this work the core metamegamodel has been extended with three concepts: annotation model, trace model, and transformation chains. In our paper the situation is more complex since our approach must be totally generic since it cannot focus on a specific domain only and since we deal with cross-layer artifacts like the weaving models linking metamodels and metametamodels. Other works on megamodeling have been developed, like the one in [25] that proposes an approach to automatically build a usable cartography of existing software platforms by merging generated metadata with user-specified metadata; in this work the authors propose a solution based on a combination of megamodeling, model transformation and model injection. Even if our work shares some technological aspects with the one proposed in [25] (i.e., megamodeling, model transformations for automation, etc.), the scope and objectives of the two works clearly differ since our proposal is generic, whereas they focus on a solution for a specific problem.

8 Conclusions and Future Work

This paper presented an approach to better understand the impact significance of the changes operated on a domain metamodel. This is considered useful for several reasons, in particular

- to realize formal documentation to be conveyed to implementors in charge of realizing the needed adaptation; and
- to estimate how diffused through the various artifacts is the impact and substantiate the efforts necessary to re-establish the validity of the compromised components.

Many approaches have been already proposed to deal with the problem of metamodel co-evolution. Each of them purported proper techniques and tools able to manage the adaptation of a specific class of modeling artifacts. Re-establishing the validity of artifacts which have been compromised by changes in the domain metamodel can be seen as a *three-steps process* (see Figure 2), whose first two steps are preparatory to the last one – the design and realization of the adaptation tools. In current practices, these steps are usually blurred and end-up mixing the impact assessment and the adaptation semantics in an intricate way. This endangers the overall consistency and can eventually lead to a more pronounced information erosion. Thus, being able to formalize the relatedness of the involved artifacts provides anyhow the designers with insights and information otherwise very difficult to elicit.

In this paper we identified the typical relations between a domain metamodel and its related modeling artifacts, and represented them with a megamodel. Megamodels have been proven to be useful in describing those infrastructures where different kind of artifacts are involved and connected together by means of mechanisms, such as model transformation and/or model weaving. Thus, a general architecture is proposed to automatically define the relation, called *Dependency*, between modeling artifact and domain metamodel. Dependencies can be derived using the characterization of the *Relation* between the meta-metamodel and the artifact metamodel. This permits to highlight those dependencies which have been affected by the domain metamodel evolution and to have an overall overview of the change propagation. A suitable megamodel describes the whole ecosystem and weaving models formalizes the relationships between the artifacts and the domain metamodel. The technique is general and, as aforementioned, provides tools independently from the chosen adaptation approach.

Future works are related to the possibility to investigate how to increase the degree of automation in the adaptation by using the outcome of the process presented here, i.e., how to use the megamodels or a fragment of them to improve the kind of adaptations which can be done in an automated way. The automation will always be partial but can be enhanced by using the knowledge encoded in the weaving models of the megamodel. On a different perspective, we are interested in investigating how to parametrize current modeling platforms in order to give the modeller the possibility to have *user-defined* relationships, specifically the conformance relation. This could be useful in defining a model-driven development process which could take advantage of the flexibility of having ad-hoc typing constraints depending on the stage of the development process. Finally, as anticipated in Section 6, we are evaluating to work on enhancing the graphical user interface exposed by our approach; for example, a possible solution could be to build on and adapt the visualization tool for traceability links presented in [34].

References

[1] Freddy Allilaire, Jean Bzivin, Hugo Brunelire, and Frdric Jouault. Global Model Management In Eclipse GMT/AM3. In *ECOOP2006*, 2006.

[2] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[3] Mikaël Barbero, Frédéric Jouault, and Jean Bézivin. Model driven management of complex systems: Implementing the macroscope's vision. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '08, pages 277–286, Washington, DC, USA, 2008. IEEE Computer Society.

[4] K. Beck and M. Fowler. *Bad smells in code*, pages 75–88. Addison Wesley, 1999.

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

[6] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture, European MDA Workshops: Foundations and Applications*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.

[7] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Procs of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop*, 2004.

[8] Jan Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 111–119, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[9] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 222–231. IEEE Computer Society, 2008.

[10] Antonio Cicchetti, Davide Di Ruscio, Dimitrios S. Kolovos, and Alfonso Pierantonio. *A test-driven approach for metamodel development*, chapter of the book Emerging Technologies for the Evolution and Maintenance of Software Models, pages 319–342. IGI Global, 2012. http://www.igi-global.com/chapter/test-driven-approach-metamodel-development/60726.

[11] M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a Generic Model Weaver. In *Procs. of IDM05*, 2005.

[12] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. What is needed for managing co-evolution in mde? In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, pages 30–38. ACM, 2011.

[13] Davide Di Ruscio, Ralf Laemmel, and Alfonso Pierantonio. Automated co-evolution of gmf editor models. In B. Malloy, S. Staab, and M. van den Brand, editors, *3rd International Conference on Software Language Engineering (SLE 2010)*, number 6563 in LNCS, pages 143–162. Springer, Heidelberg, October 2010.

[14] Eclipse project. GMF - Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.

[15] EMFText project. Emftext: concrete syntax mapper. Available on line at `http://www.reuseware.org/index.php/EMFText`.

[16] Romina Eramo, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. A model-driven approach to automate the propagation of changes among architecture description languages. *Software and Systems Modeling*, (To appear, available online ¡a href="http://dx.doi.org/10.1007/s10270-010-0170-z" target =blank¿ here ¡/a¿), 2010. special theme on Model-Based Interoperability.

[17] Jean-Marie Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Procs. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM 2003*, Amsterdam, September 2003.

[18] Mathias Fritzsche, Hugo Bruneliere, Bert Vanhooff, Yolande Berbers, Frédéric Jouault, and Wasif Gilani. Applying megamodelling to model driven performance engineering. In *ECBS '09*, pages 244–253, 2009.

[19] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. pages 52–76, 2009.

[20] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[21] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *FMOODS'06*, volume 4037 of *LNCS*, pages 171–185. Springer-Verlag, 2006.

[22] Frédéric Jouault, Bert Vanhooff, Hugo Bruneliere, Guillaume Doux, Yolande Berbers, and Jean Bezivin. Inter-dsl coordination support by combining megamodeling and model weaving. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2011–2018, New York, NY, USA, 2010. ACM.

[23] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2:176–201, April 1993.

[24] T. Levendovszky, D. Balasubramanian, A. Narayanan, and G. Karsai. A novel approach to semi-automated evolution of dsml model transformation. In *Second International Conference on Software Language Engineering, SLE 2009, LNCS*, volume 5969, Denver, CO, 05/2010 2010. Springer, Springer.

[25] V. Mahé, F. Jouault, and H. Bruneliere. Megamodeling Software Platforms: Automated Discovery of Usable Cartography from Available Metadata. *Reverse Engineering Models from Software Artifacts (REM 2009)*, page 29, 2009.

[26] David Mńdez, Anne Etien, Alexis Muller, and Rubby Casallas. Transformation migration after metamodel evolution. In *International Workshop on Models and Evolution (Me'10) - ACM/IEEE MODELS'2010*, 2010. to appear.

[27] L. Rose, A. Etien, D. Méndez, D. Kolovos, R. Paige, and F. Polack. Comparing model-metamodel and transformation-metamodel coevolution. In *International Workshop on Models and Evolutions*, 2010.

[28] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *ICMT*, pages 184–198, 2010.

[29] Louis Mathew Rose. *Structures and Processes for Managing Model-Metamodel Co-evolution*. PhD thesis, Department of Computer Science University of York, 2011.

[30] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

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

[32] Andy Schürr, Bran Selic, Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. *Automatic Domain Model Migration to Manage Metamodel Evolution*, volume 5795, pages 706–711. Springer Berlin / Heidelberg, 2009.

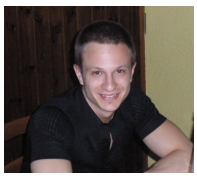
[33] Marcel van Amstel, Mark van den Brand, and Luc Engelen. An exercise in iterative domain-specific language design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 48–57, New York, NY, USA, 2010. ACM.

[34] Marcel van Amstel, Mark van den Brand, and Alexander Serebrenik. Traceability visualization in model transformations with tracevis. In Zhenjiang Hu and Juan de Lara, editors, *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 152–159. Springer Berlin / Heidelberg, 2012.

[35] Mark van den Brand, Zvezdan Protic, and Tom Verhoeff. A generic solution for syntax-driven model co-evolution. In Judith Bishop and Antonio Vallecillo, editors, *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 36–51. Springer Berlin Heidelberg, 2011.

[36] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4069 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2007.

About the authors



Ludovico Iovino is Ph.D. Student at the University of L'Aquila under the supervision of Prof. Alfonso Pierantonio. His areas of interest include model driven engineering and more specifically model differencing, and model evolution. In 2009, he received the degree in Computer Science from University of L'Aquila, working on a thesis entitled Generating java/Liferay applications from beContent models. Contact him at `ludovico.iovino@univaq.it`, or visit `http://www.di.univaq.it/ludovico.iovino`.

Alfonso Pierantonio is Associate Professor in computer science at the University of L'Aquila (Italy); he is currently director of the Master in Web Technology degree program. His current research interests include Model-Driven Engineering and in particular the theory and practice of model versioning/evolution with a specific emphasis on coupled evolution. In particular, he investigated the problem of co-evolution between metamodels and other artifacts in order to define the basis for their (semi) automatic adaptation. He has been and is currently part of program and organization committees of conferences and has been among the initiators and in the steering committee of the International Conference on Model Transformation (ICMT). He co-edited several special issues on Model Transformations which appeared on Science of Computer Programming and are about to appear on the Intl. Journal of Software and Systems Modeling.
Contact him at `alfonso.pierantonio@univaq.it`, or visit `http://www.di.univaq.it/alfonso`.

Ivano Malavolta is a Research Fellow in Computer Science at the University of L'Aquila. His research interests include software architecture languages (ADLs), architectural interchange and interoperability between software architecture languages. He is investigating the aforementioned research topics by studying Model-Driven Engineering techniques like: model-to-model transformation, model weaving and megamodeling. He is also a freelance developer and designer of Mobile and Web Applications. Contact him at `ivano.malavolta@univaq.it`, or visit `http://www.di.univaq.it/malavolta`.