Composing Graphical Representations of Composed Metamodels

Iván Melo
Universidad de los Andes
im.melo33@uniandes.edu.co

Mario Sánchez
Universidad de los Andes
mar-san1@uniandes.edu.co

Jorge Villalobos
Universidad de los Andes
jvillalo@uniandes.edu.co

Abstract

Metamodel composition is a commonly used technique in several domains that allows reusing existing metamodels and reducing the effort and time of development in new projects. However, there is a problem when in these contexts there is a need of creating new graphical editors with customs graphical representation to these composed metamodels, due to the fact that in each new project the editor must be created from scratch. To solve this problem in this paper we propose to express the graphical representations of those metamodels as independent model in order to compose them in a similar way of how metamodels are composed. In order to archive such composition we propose a set of languages and other mechanisms to deal with these compositions and the meaning of them in the context of graphical representation. Finally we introduce our proposal from a perspective in the context of Enterprise Architecture.

Categories and Subject Descriptors CR-number [subcategory]: third-level

General Terms term1, term2

Keywords Metamodeling, metamodel composition, metamodel adaptation, automatic tool generation, graphical representation, graphical composition

1. Introduction

Today, in many different contexts is common to use modeling languages that allow constructing specialized diagrams. In these contexts, the modularization of concerns is a widely use mechanism since it allows to separated concerns into different metamodels and then, reuse them by composition creating a new metamodel that fits the scope and concern of the project saving time and effort. In a preview paper [1] we proposed the language EnAr-Fusion as a possible mechanism to build a metamodel by using operations of composition and adaptation.

However, creating a composed metamodel is just part of the problem since is almost necessary to create a graphical editor to create models conform to the metamodel. In these cases the modelling languages may require a particular visual representation when the generic notation provided by the tools is not the adequate, for example when there is an established standard or when it is desirable to have several visual representations that fit the needs of different users.

Nevertheless in EnAr-Fusion as in other tools that allow the composition of metamodels, the visual representation of metamodels has not been considered. This means you cannot easily create a visual representation for a composed metamodel, based on the existing visual representation of the originals metamodels. So it would be interesting to modularize the construction of editors by using the same approach use in the construction of metamodels in the construction of the graphical representations for those metamodels. This approach would allow reusing existing graphical representations in new projects and the easy generation of editors for composed metamodels which allows having tools that fits the specific needs and representations of each project.

In today Industry there are several tools that help in the creation of editors, by using a strategy that separate the description of the metamodel from the visual representation. This independence facilitates the evolution of each of the parties and the adaptation to new requirements that may affect either the graphical representation or the structure of the language itself.

Our proposal use the separation of concerns (visual representation and metamodel) of others proposal but also introduces the idea of visual representation composition to facilitated the reuse of existing representations and make easier and faster the process of creating a visual editor. In this case the process would have four additional steps. In the first one, for each metamodel to be involved in the composition, we must select the graphical representation we want to use, which could be an standard representation or another choose from a set of available representations for each metamodel. Then, for each metamodel involved, we explain how to adapt its graphical representation. Then we must be able to express the way of composing the resulting graphical rep-
representations, in a way that reflects the desired metamodel composition. Finally, we generate the model manipulation tool. This idea is illustrated in Figure 1, where two metamodels and a graphical representation for each of them are selected from repositories. Then each metamodel and graphical representation is adapted and then composed to fits the scope and needs of a specific project.

![Figure 1. how to build the metamodel M12' and a Graphical Representation GR-M12' as a composition (C and GC) of the adaptations (A and GA) of metamodels M1, M2 and their graphical representations](image)

In this paper we present 3 things: (1) a language to express a graphical representation, based on the current technologies, but adjusted to solve the challenges related to adaptation and composition; (2) a language to compose the graphical representations in line with the composition of metamodels. This language is based on our proposed language called EnAr-Fusion [1] Even though the same ideas can be applied to any other language of the same kind. (3) The infrastructure implementation necessary to use the proposed approximation.

Finally we illustrated our proposal in the context of Enterprise Architecture projects, where there are extensive catalogues of metamodels, each of them representing a particular aspect of the problem: business processes, organization structure, strategic model, IT infrastructure, etc. These metamodels are not necessarily disjoint and may represent similar concepts even when it is usual each of them reflect a different standard: ARMOR [9], BMM [10], ArchiMate [8], BPMN [11], KAOS [12], and others. So at the time of starting an enterprise architecture project, the architect must select the domains that are inside the scope of the project and then must adapt and compose them until he reaches a metamodel that reflects the structure of the models to be built. Then he would need an editor that helps him and analysts express the current or desire state of the organization through the models.

The paper is structured as follows: Section 2 provides a summary of the composition of metamodels, the expression of a graphic representation and the tools to generate tools. Section 3 presents explicitly the requirements and challenges faced addressing the problem presented in the Introduction. Section 4 explains the solution built by us. Section 5 shows the main conclusions of the paper.

2. Meta-model composition and tool generation

Up next we present a review of the topics of interest to this proposal as are the current tools for model composition and the tools currently use to generate graphical editors of specific graphics syntax.

2.1 EnAr-Fusion: a metamodel composition language

The construction of other metamodels by reusing existing ones allows saving time and effort in the process. This reuse include two big operations: adaptation which is a transformation that allows adding, modifying or deleting elements of the base metamodel and composition that permits to join two or more metamodels. So a language to adapt and compose metamodels is a syntax through which you can express the transformations that must be applied to the base metamodels to obtain the desire one.

Nowadays there are different languages and tools to perform transformations on models. Among the most popular tools is Atlas Model Weaver (AMW) [5]. Tool that allows by means of a language known as ATL to express relations between concepts in different models, so that then can be woven into a new compound model. Another tool is Kompose [7], which has a different approach from (AMW), since the relations are inferred by the tool through the use of signatures, which can be defined as a set of properties through which you can determine whether two different concepts can be composed, otherwise they are simply added to the resulting model.

As part of our proposal we need to use a composition engine to compose the metamodels to be used in a grafical syntax. So we had chosen EnAr-Fusion[1] due to its simplicity and the fact that unlike Kompose we can specify each composition. Also its language is easier than ATL, and allows specifying the composition from a higher level. Nevertheless it could be replaced with any other composition framework without changing the result.

EnAr-Fusion [1] enables through a single process of transformation the adaptation of metamodels, and the composition of composed metamodels. To accomplish this, EnAr-Fusion offers seventeen operations that can act either on the initial metamodel metamodels as on the composed metamodels. These operations are summarized in Table 2.1
and illustrated with the two metamodels showed in Figure 2.1. We assume that the packages containing Ecore metamodels are called exampleM1 and exampleM2.

Figure 2. The metamodels which will be compose in different ways through the EnAr-Fusion operations

A composition program based on EnAr-Fusion is formed by a sequence of operations\(^1\) that must be applied in order on the metamodel composed. The operations mentioned above are implemented in an ATL-based engine.

2.2 Graphical languages and tool generation

With regard to a textual language, graphic language offers several benefits like that they are more concise, communicatively and easy to understand by stake holders \([13][14]\). The drawback is that they require much more complicated tools to develop, while a textual language can be used even with a plain text editor, a graphical language cannot be used unless you have a specialized editor. That is why in recent years, had emerged several tools that attempt to facilitate the creation of editors and interpreters in the same way in past decades emerged tools for textual languages. MegaEdit+ is one of the most advanced tools in this area. It is a proprietary tool and his utility had been proof in many industrial applications \([15]\). Among its most important features is that it allows an easy definition of advance graphical editors for modelling conform to a metamodel defined in the tool (in a proprietary syntax) through the use of dialogues, editors, languages and restrictions editors. Furthermore it provides a repository to store all the models and modify them as there are changes in the metamodel.

Compared with MetaEdit++, most of the open tools are not as advanced in terms of the easiness to create graphical representation for elements of a metamodel. For example GME (Generic Modelling Environment) is a language that supports an advanced management of metamodels (transformations, composition, and interpretation). However with regard to the graphical representation it only supports a simple model in which each element can have an associated bitmap, or may have a “decorator” that programmatically decide how to represent the item, how to represent the connections with other elements, and where locate the points of contact \([2]\). A tool comparable to GME is PtolemyII, which is based on Diva for graphical representation of elements. Diva is a software infrastructure for visualizing and interacting with dynamic information spaces.\([16]\) The project differentiates between 3 different entities: the areas that can be seen as the editors where information is viewed and modify, the InfoSpace that can be interpreted as models in a MVC style, and data sources or “background”, which are the silos from where the information come. A special feature of this proposal is that one view can be modularized in different sections of editors.

Over the Eclipse platform there is also a set of tools and libraries that allow the description of graphic representations and the generation of tools. One of the most used is GMF \([2]\) Graphical Modeling Framework, which allows creating graphical editors for metamodels. GMF Eclipse integrates two elements. On one side is EMF -Eclipse Modeling Framework, the framework that allows to manage models and metamodels, it also supports other tools that allow working with models, for example using transformations. Then there’s GEF - Graphical Editing Framework, a framework for creating visual graph-based editors. This framework provides facilities for handling the graphical representation of the elements and to handle the user’s interaction with graphs.

GMF was created to integrate EMF and GEF, allowing GEF based editors assist to build models conform to EMF metamodels. In addition, GMF provides the necessary elements for the construction of the editors to be as automatic as possible and to maximize the advantages offered by EMF and GEF: So while the first framework is responsible for everything that has to do with consistency and the persistence of models, the second one offers a high flexibility to describe graphical representations and to manage interactions. To generate a graphical editor for a new language using GMF is necessary to have the following artefacts. First, you must have the EMF model that describes the abstract syntax of the language (the metamodel for the language). When the editor is completed the models to be built using it will be conform to this metamodel. Then it is necessary to have a set of files describing the characteristics that are required in the editor. The first file is known as the GMFGraphModel
**Link** Establish a new relationship between two entities. Among the attributes of a relation are its id, cardinality, if it is a relation of composition and the inverse relation.

```plaintext
Link theB {
  metaEntitySource = exampleMM2.W
  metaEntityTarget = exampleMM1.B
  containment = true
  minCard = 1
  maxCard = 1
  associatedLink=theW
}
```

**MergeEntities** This operation is used to create a new entity from 2 existing entities. The new entity is a mixture of the two previous entities.

```plaintext
MergeEntities {
  metaEntity1 exampleMM1.C
  metaEntity2 exampleMM2.X
  NewEntity CX
}
```

**SplitEntity** Creates 2 new features from an existing one. Attributes and references of the original entity must by distribute among the new created entities

```plaintext
SplitEntity exMM2.X {
  NewEntity X1
  Attributes {
    Description theW
    AttX
  }
  References {
    theZ
    Name
  }
}
```

**newEntity** Creates a new entity in the composed metamodel. This entity will not have attributes or relations with others elements.

```plaintext
NewEntity E
```

**DeleteEntity** Deletes an entity to avoid its appearing in the composed metamodel.

```plaintext
DeleteEntity exampleMM1.B
```

---

### Table 1. Some of the instructions available in EnAr-Fusion

However as presented in [2], the fact that exist so many artefacts to describe a graphical representation makes the description error prone and very difficult to administrate towards change. Due to the fact that for any modification in the metamodel all models of the graphical description must be reviewed.

To fix this, Eugenia tool was developed [2]. This tool unifies the description of the metamodel with the description of graphic representations by using annotations over the metamodel to describe the way the elements must be painted. By doing this the complexity of the graphical description and the administration of change is reduced. However this approach presents two important trades-offs. The first one in...
exchange to reduce the number of artefacts to administrate and facilitate the evolution, the metamodel must be polluted with annotations that have no relevance in its domain. Furthermore, Eugenia increases the level of abstraction of GMF to facilitate the creation of new graphical editors, this means that the frameworks is less expressive than GMF and allows less customization. Also it is built on GMF for this reason to generate code requires the same models that GMF but in this case these are generated automatically by a plug-in that interpreted the annotations included in the metamodel.

Another existing language of graphical description is GenGMF which like Eugenia is a tool built over GMF and like Eugenia seeks to reduce the difficulty of describing graphic representations. Their premise is that in the graphic descriptions of GMF there are many graphic descriptions that despite being equal in most of the concepts have to be written many times. That is the reason why they propose to include all these concepts in a model call genGMF. In this way they reduce the number of models used to describe a graphical representation to only two, reducing the complexity of GMF.

3. Requirements and challenges in an abstract solution

In this section we first establish the objectives and general requirements of our proposal and then we explained it, through a general process the steps we propose to solve the problem addressed during the paper.

3.1 Objectives of the proposal

Our proposal aims to solve the problems related with defining and creating editors for graphical syntaxes through the modularization of the problem. In concrete we want to be able to generated editors as advance as possible for specialized modelling languages and to store the definition of those languages and those tools (editors) in deposit so we can then reuse it in future projects or during the evolution of those languages.

To achieve this goal we think is important to separate the concerns involved in the construction of graphical editors into different users with expertise in those concerns. An to do this we propose to separated the definition of the syntax from the way it should be painted, and store both files in deposits. Since the graphical representation is separated from the definition of the language it should be possible to have many graphical representations for each metamodel (language definition) and to reuse an existing graphical representation in others metamodels. This implies there must be an easy way to relate elements of the graphic language with elements of the modelling language.

Finally, the use of metamodels composition must be a guide to express the composition of graphical representation, which means it should be possible to select some of these languages and graphical representations adapt and compose them, forming a new graphical language.

3.2 A Generic Process for composing graphical representations

Up next we describe how we propose should be the process for generating a graphical editor for a project using a set of generic tools.

To illustrate the process we suppose there is a repository with metamodels from many domains and graphical descriptions for each of them.

During the first stage (S1), the first user to intervene is the metamodeler and his mission is to select from a common repository (where all users store their contributions) metamodels that describes some domains of interest to the project. Then he must use some tools to express how he wants to adapt and compose the selected metamodels in order to create the metamodel of the project. Finally, the metamodel user should with the help of some tool create the desire metamodel and store back in the repository the new metamodel and the description used to created them, in order to contribute to futures projects.
means to describe in some way how he wants to adapt and compose the representations of the originals metamodels in order to create a new one that fits the metamodel created by the meta-modeler. After the user has the description he should use a tool capable of composing such descriptions and returning a description of the graphical representation for the new metamodel. Finally this user also has the responsibility of store the new graphical representation and description of the graphical composition in the repository in order to be used in futures projects.

After the metamodel and at least one graphical representation of it is store in the repository. A third user, the tool manager enters and his responsibility is to produce the final editor with the help of a tool capable of receiving a metamodel and a graphical representation and return a Model of the desire Editor specific for some generating framework. Then this user can edit the model to tune the model and finally use the generating framework to create the model editor and conclude Stage 3 (S3).

Finally a modeller can use the editor to model, Stage 4(S4).

4. The proposed concrete solution

To carry out the strategy presented in our proposal we implemented a series of elements that allow us to meet the objectives listed in section 3. In Figure 8. We present the elements of our proposal and we will explain them through a simple example following the process proposed in section 3.

4.1 A Megamodel to store all the elements

The first element of our proposal is a mega model which is a big repository where we want to want to keep all the artefacts of our proposal, which include metamodels, artefacts like the graphical representations (EnAr-Picture), EnAr-Fusion and EnAr-Gromp (our language to describe adaptation and composition of the graphical representations). In this megamodel there is a place to store metamodels, another to store description of graphical representations and another to store the artefacts mentioned above. An important characteristic of this megamodel is that allows storing relations between the artefacts and metamodels involved. Following the process described in section 3, a meta-modeller user start the process by selecting from the repository the metamodels he thinks would help to create the new graphical syntax. For our example let's assume the user selected metamodels M1 and M2, presented in Figure 9.

During the next step of the process the meta-modeller user writes an EnAr-Fusion [1] script expressing how he wants to adapt and compose the metamodels selected during the first step. In figure 10 we show the resulting metamodel M12’ (MM12’ = C(( A(MM1), A(MM2)) ). Finally the meta-modeller user summit the metamodel MM12 and the EnAr-Fusion to the megamodel so it can be used by others. Figure 10 shows the composed metamodel.

During the next step of the process the meta-modeller user writes an EnAr-Fusion [1] script expressing how he wants to adapt and compose the metamodels selected during the first step. In figure 10 we show the resulting metamodel M12’ (MM12’ = C(( A(MM1), A(MM2)) ). Finally the meta-modeller user summit the metamodel MM12 and the EnAr-Fusion to the megamodel so it can be used by others. Figure 10 shows the composed metamodel.

4.2 Expressing the graphical representation - EnAr-Picture

During stage S2 the user in charge of composing the graphical representations selects one for each metamodel. These
Figure 9. Metamodel M1 and M2 of our example, the first one represent the structure of an organization while the second Business Process Architecture.

For the construction of EnAr-Picture we assume the main concern about graphical representation is to describe how to paint the concepts of elements of a metamodel and the relations among them although EnAr-Picture also allows representing a concept as a relation. So in order to describe the graphical representation our language has 3 main characteristics: Node (concept), AttLink (relation), NodeLine (concepts as relation). Around these concepts we create some entities to express in an easy way the variants of the concepts.

In Figure 11 we present the syntax to import a metamodel, create the main concept of the graphical representation and express what concept is the one that contains the others.

Figure 11. Script to Import a Metamodel and start the description of the graphical Representation

4.2.1 Types of Nodes

Even though in general every model has nodes for the graphical representation we have specialized this concept into three different types of possible representation, since in an editor there are three ways to represent a concept (as an image, with the default representations or as a custom representation).

For our example we create a graphical representation for the metamodel of Organizational structure and choose to represent an Organizational Unit and Role with default representations, and introduce an image to represent a person. In figure 12 we present how to express this with EnAr-Picture.

Figure 12. Script to create the graphical representation of the concepts of metamodel M1, and how it could look
4.2.2 Types of Relations

EnAr-Picture allows to easily described the relations among different concepts with a simple description, and provided three different ways to paint the relations: as one node inside the other, as one node around the other or with some type of line between the two concepts.

In Figure 13 we show how to describe such relations with our example of the Organizational Structure metamodel, expressing the relation between an Organization Unit and a Role as one node inside the other, and creating some lines to express the relations among the other concepts of the metamodel.

Figure 13. Script to create the graphical representation of the concepts of metamodel M1, and how it could look

4.2.3 Additional concepts

EnAr-Picture has 23 concepts that allow expressing many variations around the graphical representation, such as types of lines, color, layouts, etc. It also provides some extension point that helps include new characteristics to a graphical representation. In Figure 14 we present the metamodel of EnAr-Picture.

Finally in Figure 15 we present the Editor created with metamodel M1 and the script presented through the section.

4.3 Graphical Composition Language - EnAr-Gromp

The second language of our proposal is called EnAr-Gromp and was developed to describe the operations that can be made over graphical representations in order to compose them. This is because graphic operations do not work like compositions over metamodels, due to the fact that instead of established relations between different concepts, what they allow is to give meaning to those concepts in a visual level.

EnAr-Gromp is an imperative language that extends EnAr-Picture and allows composing Graphical representations using a simple set of instructions hiding the complexity and extension of a graphical representation, reducing the time to create a new picture script. To do so, EnAr-Picture has an instruction to import the source metamodels, graphical representations, target metamodel, and Fusion script.

4.3.1 Operations Over Nodes

A second set of instructions of Enar-Picture aims to operate over the nodes allowing to alter graphical representation, changing the concept it represent or some characteristic of the same, in Table 2 we present what those instruction over nodes means, over some random graphical representation for any concept.

The first three operations Join Side Order, Merge, Or and Replace are used when in EnAr-Fusion or any other language you merge two entities and you have to decide what graphical representation leave since you have two different graphical representation. With join side order you just create a new graphical Representation that has the 2 previously created. With the Merge operation you merge the two graphical representations by overlapping them. The Or Operation
4.3.2 Operations Over Relations

The third set of instructions of EnAr-Gromp operates over Relations, their intention is to create in a simple way the same relations that can be declare with EnAr-Picture but among nodes of different metamodels. In this set there is also another instruction to delete a graphical representation for relations by using the instruction Unrelated. In Table 3 we introduce them in a graphical way.

4.3.3 EnAr-Gromp Example

In this section we present a brief example of how through the use of EnAr-Gromp Graphical representations can be reused in new projects saving time by just reusing and adapting already existing projects.

First lets assume we have the Graphical representation for metamodel M1 presented during last section. Also we have a graphical description for metamodel M2 that can be seen in Figure 16. So now we want to create a new Graphical representation for the composed metamodel created during the first step of our process and presented a few section ago in figure 10.

Table 2. Instructions available for nodes in EnAr-Gromp

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join Side Order</td>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>Merge</td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
<tr>
<td>OR</td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td>Replace</td>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
<tr>
<td>Create</td>
<td><img src="image9.png" alt="Image" /></td>
<td><img src="image10.png" alt="Image" /></td>
</tr>
<tr>
<td>Change</td>
<td><img src="image11.png" alt="Image" /></td>
<td><img src="image12.png" alt="Image" /></td>
</tr>
<tr>
<td>Split</td>
<td><img src="image13.png" alt="Image" /></td>
<td><img src="image14.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Table 3. Instructions available to link nodes in EnAr-Gromp

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>contain Outside Position</td>
<td><img src="image15.png" alt="Image" /></td>
<td><img src="image16.png" alt="Image" /></td>
</tr>
<tr>
<td>contain Inside Layout</td>
<td><img src="image17.png" alt="Image" /></td>
<td><img src="image18.png" alt="Image" /></td>
</tr>
<tr>
<td>toLink</td>
<td><img src="image19.png" alt="Image" /></td>
<td><img src="image20.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Figure 16. The final editor for metamodel M2
Since our goal is to create a graphical representation for metamodel MM12 and we will be using the existing graphical representations all we have to do is to choose a graphical representation for the entities that exist in both metamodels. In our example we will just choose the graphical representations of metamodel M1, since are nicer. So our script will be really simple, first we import the metamodel M1, M2 and the previously composed M12, and the graphical representation for metamodels M1 and M2.

```
import "platform:/resource/Example/model/organizationalStructure.ecore" as M1
import "platform:/resource/Example/model/BPA.ecore" as M2
import "platform:/resource/Example/model/MM12.ecore" as MC
import "platform:/resource/Example/scripts/MM12.fusion" as fusion
import "platform:/resource/Example/GR/M1.picture" as GR1
import "platform:/resource/Example/GR/M2.picture" as GR2
```

Next we choose the graphical representation of metamodel M1 for the concept: Role. The following EnAr-gromp instruction will do the trick:

```
Or (GR1.Role, GR2.Role) -> GR1.Role
{
targetClass <- MC.Role
toolName "The composed Role"
toolDescription "Merge Role"
}
```

Next we do the same for the graphical representation of concept: Organizational Unit:

```
Or (GR1.OrganizationalUnit, GR2.OrganizationalUnit) -> GR1.OrganizationalUnit
{
targetClass <- MC.OrganizationalUnit
toolName "The composed Organizational Unit"
toolDescription ",Organizational Unit,description"
}
```

And since there are not new concepts or relations to represent among the concepts of the composed metamodel, the engine will automatically pass the rest of the graphical representation to the new file, resulting in the editor show in Figure 17.

### 4.4 Metamodel Composition Engine - EnAr-Fusion

Our proposal include three engines, the first one named Fusion is the engine behind the language EnAr-Fusion it has two main components, a textual editor that help the meta modeler to write the Metamodel composition script, an a Interpreter that load the Fusion script, load the original metamodels and creates a composed one following the instructions of the script.

As all the others engines of our proposal the EnAr-Fusion engine was built with two main frameworks. The editors were built with Xtext [17] and the Interpreters are built with the EMF framework, using dynamic EMF. One of the main advantages of this approximation is that allows us to create a plugin that can be include in Eclipse.

### 4.5 Graphical Composition Engine - GCE

The second engine named GCE (Graphical Composition Engine), is the one that handle the composition of graphical representations through the interpretation of the EnAr-Gromp script. This tool has two parts; the first one is the editor that helps the graphical modeler expert to write the script. The second one is the engine that receives the EnAr-Gromp script, and the others inputs such as: the originals metamodels, the selected graphical representation for each of them, the composed metamodel (result of EnAr-Fusion), the EnAr-Fusion script, and return a new graphical representation (expressed in in EnAr-Picture), for the composed metamodel.

### 4.6 Tool Generator Model - TGM

This engine handles the generation of the tool model. In our proposal the engine receives as input a metamodel and a description of the graphical representation (EnAr-Picture) and generates a tool model. In our context a tool model is a concrete model (and artefacts) that can be used as input to a generator of graphical editors. In our implementation the tool model generated with the engine is accordant with Eugenia so our engine generates a GMF Project that contains the composed metamodel with the annotations of Eugenia, and others files like Java classes and advance artefacts from Eugenia (like eol transformations, and plug-in projects). The principal reason why our engine creates a tool model rather than the editor itself is that it gives advance users the possibility to manually tune details before generating the editor.
5. Conclusions

However we leave as future work the creation of other high level tools that support and guide the users through the composition of graphical representations.

In this paper we presented our proposal to save time and effort in the construction of editors for graphical syntaxes by separating the concerns of defining the metamodel and how it is represented encouraging the reutilization of prior existing metamodels and graphical representations.

Also thanks to this approximation we are able to define different experts and concerns during the process of creating a new graphical editor.

As result of our proposal we present the languages EnAr-Picture and Enar-Group two DSL that fits the specific needs of our proposal allowing composing graphical representation in a similar way as metamodels are composed but with the clear intention of give meaning in a graphical level. We also present our framework that includes the tools mentioned in this paper that will help created editors that fits the specific needs of a project.

We leave as future work to introduce and the used of graphical representations in new contexts solving the problem of visualization when in an editor you have to work with a big number of entities and relations at the same time.

References


