Extending EMF for modularity

van Schuylenburg, S.B.

Award date:
2016

Extending EMF for Modularity

Stef van Schuylenburg

February 18, 2016

Abstract

The Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools, based on data models. Using EMF, developers can create a model specification and EMF can generate software tools from this model. Those tools allow other developers to create and work with the models in various ways. Unfortunately, the models can become very complex and difficult to maintain.

In this thesis we present a solution to handle complex models. With our solution we extend any type of model with modularity. Modularity is achieved by introducing a facility to support the use of parameterized functions, where the functions return model fragments. Each function defines a parameterized module; they contain a definition and can be called from within a model. The models with functions can be transformed to models without functions, such that the meaning of the model is preserved. Using this solution we expect to improve the maintainability of complex models that are created using EMF.

Contents

Chapter 1

Introduction

Model Driven Engineering (*MDE*) is a software engineering approach that uses models to assist in the creation of a software product. It gives developers the ability to decribe parts of the software product in models and to create resources that become part of the end product.

The Eclipse Modeling Framework (*EMF*) is a framework that supports the MDE approach. It allows developers to create domain-specific languages by generating building tools. With those building tools, developers can create models that belong to the created domain-specific language. Furthermore the models can also be used to generate code or to generate other models.

EMF does, however, not provide developers with a way to achieve modularity in the models. It is up to the creator of the domain-specific language to introduce modularity for the models, by introducing modularity concepts in the metamodel. Modularity does have many advantages. The advantages are among others [1]:

1. Changeability: The ability to change one part of the model and affecting the other parts as little as possible.
2. Independent development: The ability to work on different parts of the model at the same time by different developers.
3. Comprehensibility: The ability to understand a part of the model without having to understand the complete model first.

This thesis is structured as follows: In the next chapter we will look at MDE in more detail. In Chapter 3 we will give an overview of our solution and in Chapters 4 through 6 we will explain our solution. In Chapter 7 we will show and motivate a use case for our solution. In Chapter 8 we will look at the related work concerning modularity when working with MDE and EMF. In the final chapter we will give a conclusion and discuss the possible future work. But first we will start with introducing our research goals.

1.1 Research Goals

In this paper we present a solution to introduce the advantages of modularity to models, with the solution being exemplified on EMF models. In order to measure whether modularity is achieved, we use the following research goals:

1. Improve *changeability* of model parts.
2. Improve *comprehensibility* of the models.
3. Add *reusability* of models.

We will compare how creating models with our solution will satisfy those goals compared to creating models without our solution.

Besides being a theoretical solution, we also implement our solution. The implementation will serve both as a proof of concept and as a tool that can be used in building models.

Chapter 2

Model Driven Engineering

In this chapter we will look at MDE in more detail and we will explain the terms used in this paper. More information about the terms introduced in this chapter can be found in the Meta Object Facility specifications [2] that introduces these terms in a more formal way.

2.1 Models

MDE makes heavy use of *models*. Models are used to describe data, processes and other resources. In MDE, the term model is used in an abstract way. A model can be a piece of code, a diagram or any other kind of structured data. The model is used as an abstract representation. A model consists of *model elements* with *relationships*. *Model elements* are the objects that live in a model. In MDE, model elements often have a certain type. Model elements can have relationships to other model elements. Those relationships indicate that the model elements are related in a certain way. How they are related is indicated by the type of the relationship. Furthermore, model elements can have *attributes*. Attributes are properties of model elements that are represented in a primitive type, or another type that is defined as a data type within the metamodel. For example a model element can have properties with a number value or a string value.

An example of a model can be found in Figure 2.1. In this model we find four model elements. One of the model elements is of type Book and the others are of type Chapter. The Book element has relationships to the Chapter elements, for this example the relationships indicate that the Book consists of those Chapters. Furthermore the Book element has two attributes: one authors attribute and one title attribute. In this case, the attributes have a string value. This figure is just one way of representing the model; the model could for example also be represented as text or using a different graphical notation.

2.2 Metamodels

To define how the models are structured, *metamodels* are used. Metamodels are also models. A metamodels defines a language to which the models must adhere. Models that adhere to a metamodel, are called instances of that metamodel. To

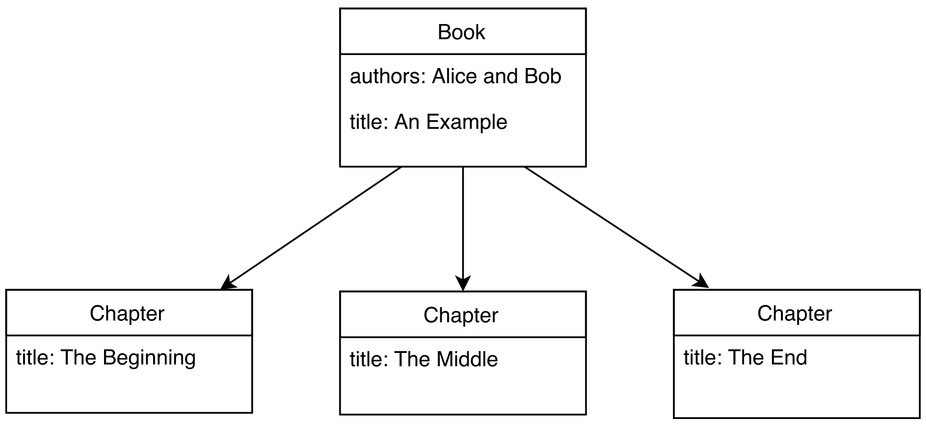

Figure 2.1: An example of a model describing a book

define the language, the metamodel specifies a number of *classes*. Those classes specify the types that are used by model elements within the instances of the metamodel. They specify the relationships and the attributes that the model elements of that type have. A language defined by a metamodel is called a *Domain-Specific Language* (DSL).

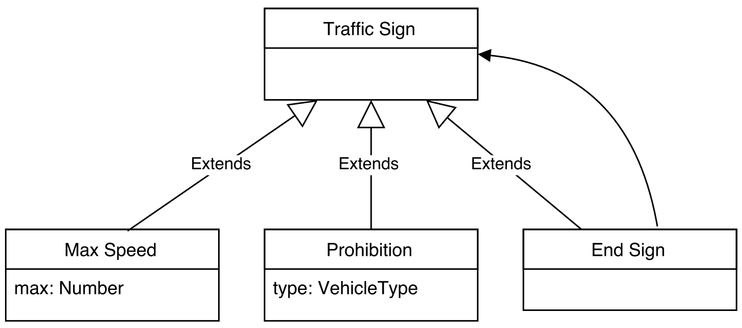

Figure 2.2: An example metamodel of the traffic signs.

To explain metamodels in more detail, we will look at an example. For this example we will look at a real world model, that is often encountered: The traffic signs. Traffic signs are models that describe the rules for the road they are located at. There are many types of traffic signs: some specify the maximum speed that is allowed on the road, others prohibit certain vehicle types to enter the road and there are traffic signs that indicate that a previous traffic signs no longer applies. The traffic signs can be seen as a type of models, which can be described by a metamodel. A small metamodel covering the traffic signs can be seen in Figure 2.2. This metamodel only covers the traffic signs which we mentioned above. In the metamodel we find one element named Traffic Sign. The other elements are classes that extend the Traffic Sign class. Two of those classes contain an attribute; Max Speed has a *max* attribute and Prohibition has a *type* attribute. The End Sign has a relationship to Traffic Sign, in this case the relationship indicates that the Traffic Sign no longer applies. Using this

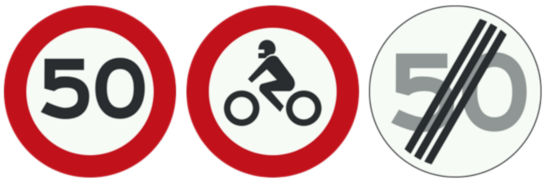

Figure 2.3: Traffic sign models

metamodel we can describe traffic signs. For example, when we have a model instance that represents a maximum speed limit of 50, then our model consists of a model element of type Max Speed, with an attribute max with a value of 50. When we have a model that indicates the end of a maximum speed limit of 50, then our model consists of a model element of type End Sign, with a relationship to a model element of type Max Speed.

Based on this metamodel, the traffic signs found in Figure 2.3 are model instances. This figure uses the representation of traffic signs as they are found in Europe.

2.3 Model Transformations

Model transformations are used within MDE to generate resources based on models. Model transformations can be created as an automated process. As input they take model instances of a certain metamodel and they can generate either other models, or text-based files. The model transformations that generate models are called model-to-model transformations. The model transformations that generate text-based files are called model-to-text transformations.

Model-to-text transformations are often used to generate code based on a model. For example, EMF generates model editors based on a metamodel. To generate the editor they created a model transformation. This model transformation takes as input a metamodel (which is a model itself) and generates the code that is used to implement the model editor.

As an example for the model-to-model transformation, we could have a transformation based on the traffic signs metamodel from Figure 2.2. Suppose we want a transformation that transforms signs into end signs and end signs into normal signs. To solve this, we could create a model transformation that looks at whether the model consists of an End Sign model element or not. If it is not an End Sign, a new model is generated consisting of an End Sign model element, with a relationship to root element of the given model. If it is an End Sign, a different model is generated consisting of the model element to which the End Sign has a relationship. Figure 2.4 shows an example of transforming a traffic sign model.

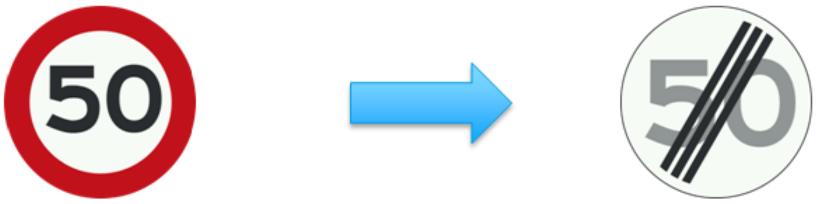

Figure 2.4: A model transformation applied to a traffic sign

Chapter 3

Overview of Our Approach

In this chapter we give an overview of our solution. First we explain how our solution is used; we explain the process of using the new concepts in models. Then we look at how we achieve our solution.

For our solution, we introduce function structures to models. A function acts as a model part, which can be used in other models and which supports the use of parameters. To use a function in a model, you need a function application within your model. A function application is a model element that indicates that we want to use the model part that belongs to the function. The function application contains a reference to the function and it may contain arguments which are used to apply the function. The function itself defines a model part. Within the model part, the function can refer to the parameters of the function, to indicate that an argument will be used there.

Our functions are placed in one or more model files separated from our main model. The functions are bundled in *Library* models which contain multiple *Function* elements. The functions are defined by a reference to the model elements that represent the model part. The elements of the model part are elements coming from the metamodel, or can be references to other functions themself. Furthermore we also allow *Param* elements. Those elements indicate that the element refers to the argument that is given at the function application. Those Param elements provide the function structure for our submodels. To create a function application, the *FunctionApply* elements are used. These elements contain a reference to the Function element. The FunctionApply elements can also contain a reference to another element which they will use as argument for the Function element. Using this structure we can use a single function at multiple locations within our models, but still allows varation within the model parts we want to use.

To use the new types of elements in a model combined with the elements from an already existing metamodel we extend the metamodel. We create a new metamodel that contains the classes from a given metamodel in addition with new classes to define and to use the functions. The extended metamodel is created with a model transformation. Now EMF can generate tools for us to create and edit models that satisfy the extended metamodel. The downside of this approach is that the new models can not be used by tools that are based on the already existing metamodel. To solve this we also supply a transformation to transform models with functions to model without functions. The transfor-

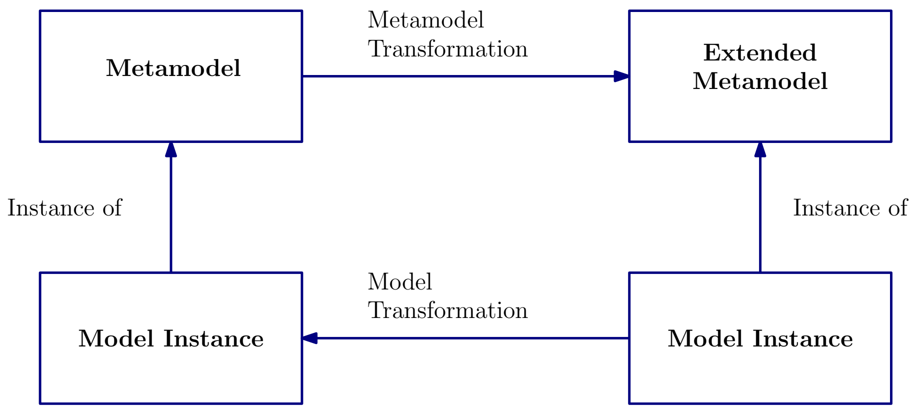

Figure 3.1: An overview of our process to introduce modularity to models.

mation is created such that the generated models without functions preserves the same meaning as the models with functions; when we apply a function in a model, the function application will be replaced by the submodel defined by the function. An overview of this process with the extended metamodel can be found in Figure 3.1. In this figure we find the transformations that are required for our solution.

Our solution works for models based on any EMF metamodel. We will start however with introducing our solution based on only one metamodel in Chapter 4 and Chapter 5. Then we introduce a way to apply our solution to any metamodel in Chapter 6.

Chapter 4

Extending Metamodels

In Figure 3.1 we presented the *extended metamodel*. This extended metamodel is based on an arbitrary metamodel and adds the concept of functions to it. We have chosen to create a new metamodel, such that we can use the code generation facility of EMF to generate the building tools to work with the extended metamodel. The building tools generated by EMF allow us to use the EMF-based model structures, which is supported by many tools that are created for EMF.

In this chapter we will show a first step to the extended metamodel. We introduce an example metamodel which we extend with the notion of functions. In this chapter our function system is metamodel dependent; the two transformations from Figure 3.1 are defined specifically for this specific metamodel. Using this example we introduce and motivate the concepts that are used for our solution.

4.1 Introducing the Metamodel

For the metamodel to extend we choose a metamodel which defines a modeling language for working with binary trees. A representation of this metamodel can be found in Figure 4.1. Here we find a recursive definition of a tree. *Tree* is an abstract class in our metamodel that can either be a *Node* or a *Leaf*. The Node class has two children: a left Tree and a right Tree. The Leaf class only contains one value attribute.

Using this metamodel we can create Tree models. An example of such a model can be found in Figure 4.2. This figure contains a graphical representation of a Tree model using the Node and Leaf classes.

4.2 Extending the Metamodel

Now we want to extend this metamodel with concepts to use functions. The functions will be model elements on the model level that take another model element as an argument. Those functions have a return type, which is represented by having a super type relation in the metamodel. In this example the functions return Tree elements, so the function classes have a super type relation to Tree. Functions are represented as normal model elements: they are just a class with

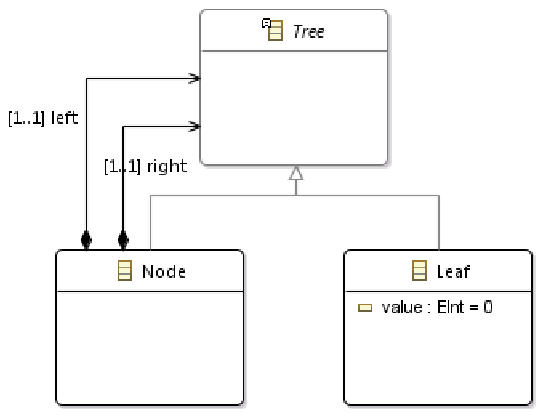

Figure 4.1: The basic Tree metamodel

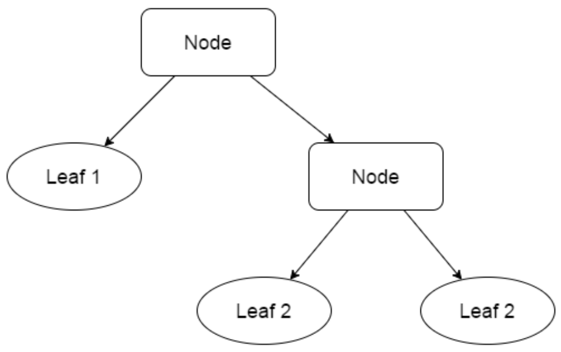

Figure 4.2: An example of a Tree model

a reference and a super type relation. The functions become special when we are transforming our instances of the extended metamodel to instances of the original metamodel. During the model transformation the function elements will be replaced by a model element. This model element may use the argument given to the function element as one of his descendants.

For this example we will create two functions. The first function is called *LeftTree*. LeftTree takes as argument a Tree and will be transformed to a new Node where the given tree is placed as the left child. The right child will be a simple Leaf model element containing value 0. We model this in our metamodel as a new class LeftTree, with a reference to a Tree and Tree as super type.

The second function is called *Duplicate*. Duplicate takes as argument a Tree and will be transformed to a Node where the children are both the given Tree. So the left child will be a copy of the right child. We model this in exactly the same way as the LeftTree. Except for the name of the classes, those two functions are modeled in exactly the same way in our new metamodel. The extended version of the metamodel can be found in Figure 4.3.

Using this extended metamodel we can create models using the Duplicate and the LeftTree functions. An example of such a model can be found in Figure 4.4. This model uses the Duplicate function, by using a model element of the

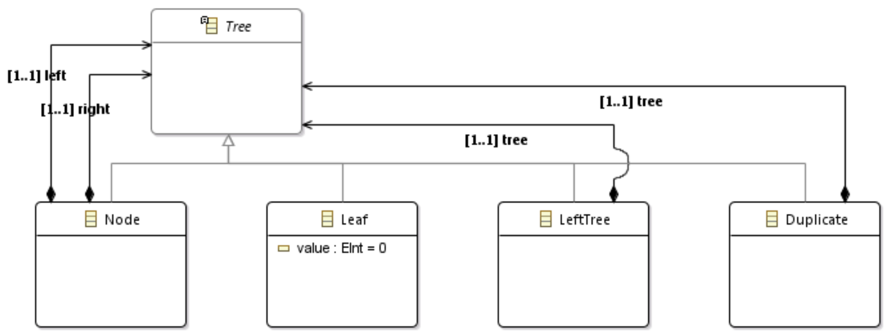

Figure 4.3: The extended Tree metamodel, containing the function classes.

type Duplicate. The Duplicate element used in this model, represents a subtree consisting of a Node with two children, which both are the Leaf 2 element as referenced to by the Duplicate element. This Tree model with functions represents the Tree model without functions from Figure 4.2. In the next section we look at how we can transform such a model with functions to a model without functions.

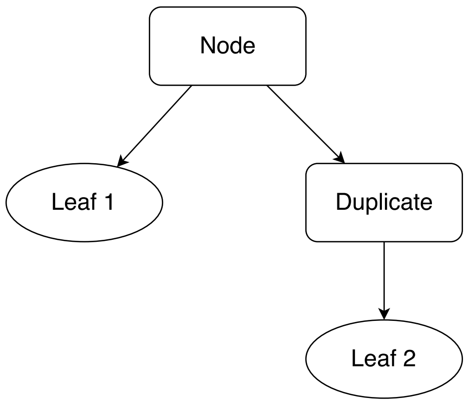

Figure 4.4: An example of a Tree model with functions

4.3 Transforming the Models

Now we want to transform instances of the extended metamodel to instances of the basic metamodel. For the transformations we use the tool QVT Operational[1]. QVT Operational is a tool that supports model-to-model transformation on the models created with EMF. For this, QVT Operational uses a DSL where the transformations are defined by mappings. Those mappings behave like functions that can be applied on model elements to transform them to new model elements.

[1]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

The transformation of LeftTree elements is defined by creating a new Node element for our target model. The right child of the Node element is a basic Leaf element. The left child is the Tree that is given as argument for the LeftTree element. To make sure that this argument satisfies the basic metamodel, we have to transform the argument to an element of the basic metamodel. It may be the case that this argument contains a LeftTree or a Duplicate element itself, so we use the same transformation for the argument as we did for the complete model.

The transformation of Duplicate elements is defined similar to the transformation for the LeftTree elements. The difference for the Duplicate function is that instead of using a basic Leaf element for the right child, we will be using the same model element as we used for the left child. For this we take a copy of the element created by transforming the argument of the Duplicate element. We use a copy, because it may be the case that metamodels are using containement relations, which forbid referencing to a single model element more than once.

We also have to transform the Node and Leaf elements from the source model to Node and Leaf elements in the target model. These transformations are trivial: we copy all the arguments into the new Node and Leaf elements.

We have chosen for QVT Operational as language for our model-to-model transformation, because with other modeling languages we faced limitations which prevented us from using it for our solution. Especially regarding the instantiation of model elements more than once we encountered limitations, because other modeling transformation languages are more declaritive than QVT Operational. Furthermore for QVT Operational we could find more documentation about using the language than many of the other transformation languages.

4.4 Conclusion

Now we can use functions in our models to achieve higher modularity and to be able to reuse structures. However using this approach it is required for the developers to modify the metamodel to introduce new functions. Often we do not want to change the metamodel, because there are multiple parts that depend on the metamodel. For this we want to find a more general solution, which does not require the developer to modify the metamodel in order to introduce new functions.

4.4.1 Workflow

Now we will look at a summary of the workflow that is required for the solution presented in this section. In this workflow, we will look at what is required to set up an environment to work with models with functions. As starting point the workflow uses a metamodel, namely the original metamodel. For this metamodel we want to use a number of functions during the creation of model instances. The workflow is as follows:

1. Create a new metamodel, containing the classes of the original metamodel.

2. To the new metamodel add one new class for each function we want to introduce.

3. Use EMF to generate an editor for this new metamodel.
4. Create a new model transformation, to transform model instances of the original metamodel to model instances of the new metamodel.
5. For each class of the original metamodel, add a mapping that copies the model elements of that class.
6. For each of the classes created for the functions, create a mapping based on how the function should behave.

Now model developers can use the generated editor to create models with functions and use the transformation to transform models with functions to model instances of the original metamodel.

Chapter 5

Defining Functions on the Model Level

Having the type of function defined by the metamodel forces us to change the metamodel if we want to add or remove a function. Having to frequently update the metamodel has multiple disadvantages, because many parts of the tools depend on the metamodel. For example we have to recreate the editor (often this can be done automatically), the transformation may have to be updated and all the models using the metamodel may have to be updated.

By allowing the developers to add or remove functions on the model level, we expect to achieve a solution that is more flexible and that requires less effort to keep all the parts consistent.

5.1 Updating the Metamodel

To define the functions on the model we need new classes in our metamodel that support this. For this we use the classes which are already mentioned in Chapter 3. This gives us the following classes:

- FunctionApply
- Function
- Library
- Param

The *FunctionApply* class is introduced such that we no longer need classes in our metamodel that only cover one function, but such that we can have one class to refer to any function. The FunctionApply class represents the application of a function, where we use a reference to a Function definition to indicate what function should be applied. In our new metamodel FunctionApply replaces the LeftTree and Duplicate classes. In our example the FunctionApply has a supertype relation to Tree, such that we can use FunctionApply elements as children of Node elements. For other metamodels other supertype relations would be required.

Next we have the *Function* class. This class is the definition of a function on our model level. To define a function we use a normal model element. The idea is that when we apply the function using a FunctionApply element, that the FunctionApply element actually will be this model element. This model element that defines the function is referred to as the *body* of a Function element. To let the Function elements behave like we expect from functions we are using the Param class, which we explain later on in this chapter.

Our approach is to keep the Function elements in a model file, separated from the model we want to transform. This allows us to reuse the defined functions in other models. To support this we use the *Library* class. A Library element holds multiple Function elements and acts as the root element for the model holding the Function elements. We want them to hold multiple Function elements, because otherwise we would need a separate model for each function, which could quickly require a large amount of models. By using one model file for multiple Function elements we expect that we are able to keep a better overview of all our models.

Finally we have the *Param* class to allow the usage of parameters in the function definitions. For the function definitions we want to be able to use the parameter within the body of a Function. So we should be able to express that we want to place the model element belonging to the given argument at a certain location. This could be the body itself, but it should also be possible to use it as a descendant of our body. Otherwise it would only be possible to create identity functions. The *Param* elements represent the model elemenst which will be replaced by the argument given to the FunctionApply element. To be able to use the Param elements everywhere in our body, we make them a subtype of Tree.

The new updated version of the metamodel can be found in Figure 5.1.

We can now create models using this metamodel. An example of the model where we are using and creating the duplicate function can be found in Figure 5.2. Here we find on the left the Tree model defined using the new version of the metamodel. On the right we find a Library model defining a function used in the Tree model. This model expands to the model from Figure 4.2.

5.2 Transforming the Models

As explained in the previous section we now use multiple models as input. The main model is similar to the model we used as input in Chapter 4, where we have a tree that may contain FunctionApply elements. The other models should have a Library element as root. These models define all the functions we are allowed to use in the main model. The model that is generated is based on the main model, but uses the other models to define how the functions behave. The transformation only needs the main model as input model; the Function elements we are using can be derived from the FunctionApply elements that refer to them.

When we transform our input model we keep the same approach as in Chapter 4: We take the Node and Leaf elements and transform those in a trivial manner and we take the FunctionApply and transform those in a structure that belongs to the Function.
The difference is that we use the Function elements defined in the other models

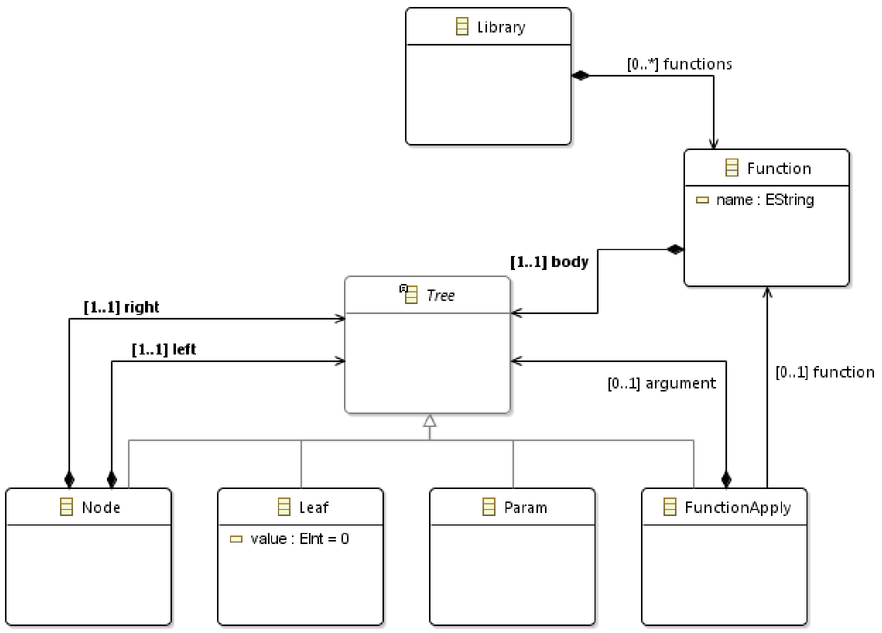

Figure 5.1: The updated metamodel, containing the more generic functions.

to transform the FunctionApply elements.

To transform the FunctionApply elements we take the body of the Function element to replace the FunctionApply. We also have to consider the arguments of the FunctionApply elements. This argument replaces the Param elements from the body of the Function element, but only for the transformation of the current FunctionApply element. We decided to achieve this by adding a parameter to the mappings of the transformation. This is a mechanism allowed by QVT Operational to execute the mappings with a parameter. When we encounter a FunctionApply element it will transform the body of the Function element with the argument of the FunctionApply element as parameter. And when we encounter a Param element we replace it by the parameter. For the other elements we pass the parameter to all the transformations of the children of the elements. We use a parameter for the transformation, because it allows us to mimic the behavior of the model in our transformation. This makes it easier to reason about it and it makes it easier to get the desired behavior. Instead of using parameters we could also use global variables. But using global variables becomes more complicated when you have a FunctionApply element within the body of a Function element. To support those you would have to create a stack-like mechanism to keep track of which argument is the current parameter.

5.3 Conclusion

Now we can edit, add and remove function definitions completely on the model level. We only have to edit the metamodel once to add the new classes in-

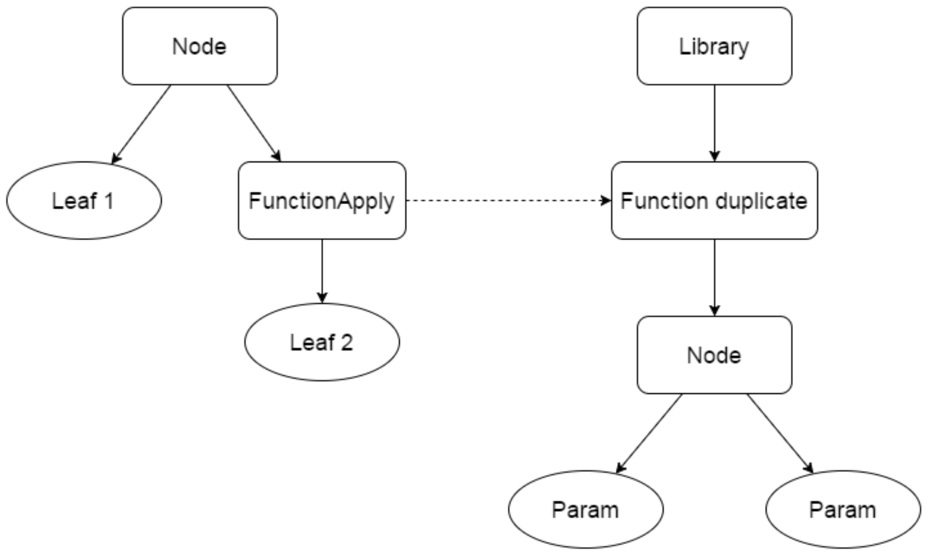

Figure 5.2: A Tree model using a function from a Library model.

troduced in this chapter. Our solution is however created specifically for our example metamodel containing Tree elements. If we want to use this approach for other metamodels we would have to edit the metamodel manually and we have to create the transformation manually. To make our solution more valuable it would be better if the new metamodel and transformation could be generated automatically. This would make our solution applicable for any kind of metamodel.

5.3.1 Workflow

Now we will look at the workflow for using the solution as presented in this chapter. We start with the original metamodel and for this metamodel we are going to set up an envorinment to work with models with functions.

1. Create a new metamodel, containing the classes of the original metamodel.
2. Add to this new metamodel a Function and a Library class.
3. For each class in the original metamodel with a reference to it, add a FunctionApply and a Param class.
4. Add to the FunctionApply and Param classes supertype relations to the class of the original metamodel, for which the classes are created.
5. Use EMF to generate an editor for this new metamodel.
6. Create a new model transformation, to transform model instances of the original metamodel to model instances of the new metamodel.
7. For each class of the original metamodel, add a mapping that copies the model elements of that class.

8. Add a mapping for the FunctionApply elements, that generates an element based on the body of the Function element, to which the FunctionApply element is referring.

9. Add a mapping for the Param class that uses the parameters that are passed through within the transformation.

And now we use the generated editor to create Library models, containing the functions we want to use in our models. Using the transformation we can take models with functions and transform them to model instances of the orginal metamodel.

Chapter 6

Automating the Procedure

Now that we can use functions in the Tree domain to achieve modularity, we like to extend this to any domain. Our current approach of extending the metamodels gives us a solution that only works for one metamodel. Extending this to another domain would require us to repeat all the steps of Chapter 5 for each metamodel. If we can automate this process, we can apply our approach to any domain with minimal effort. In Chapter 5 our solution used two files. The first file is the extension of the original metamodel, containing the function classes and the other classes to support the functions. The second file is the model transformation that takes an instance of the extended metamodel and results in an instance of the original metamodel. In this chapter we show how those files can be generated and how the generation procedure can be automated.

6.1 Generating the Metamodel

First we generate the metamodel containing the new function classes. The generated metamodel is based on an arbitrary metamodel, lets call this arbitrary metamodel the *original metamodel*. This new metamodel will be similar to the extended metamodel we created in Chapter 5 (the metamodel can be found in Figure 5.1). However, instead of using the classes Tree, Node and Leaf we use the classes of the original metamodel. The classes Param and FunctionApply do not contain a supertype relation to Tree, but to classes of the original metamodel.

To create the model to model transformation we decided to use QVT Operational. We choose QVT Operational, for the same reasons as explained in Section 4.3. In the model transformation we take the root of the original metamodel, which is an EPackage containing the classifiers of the metamodel. Using this EPackage, we create a new EPackage which becomes the root for our generated metamodel.

The classes we want for our generated metamodel can be divided into four categories: The original classes, the FunctionApply classes, the Param classes and two additional classes. Similar classes are also found in our approach for the Tree metamodel in Chapter 5. Some of the classes however are modified and some variations of the classes are introduced to be able to handle any arbitrary metamodel. We now look at the four categories and motivate why they are changed compared to the solution proposed in the previous chapter.

The complete QVT Operational transformation can be found in Appendix A.

6.1.1 The Original Classes

The original classes denote the classes that are found in the original metamodel. All the original classes must also be included in the extended metamodel, because they define the domain we want to extent.

We have two options to include for the original classes: we can refer to the classes in the original metamodel from our generated metamodel or we can copy the classes of the original metamodel into the generated metamodel. The first solution would keep the generated metamodel smaller and keep the original classes separated from the classes belonging to our extension. However referring from one metamodel to another is not that well supported by the tools. For example QVT Operational does not allow a mapping consisting of a disjunct between classes of different modeltypes. For our proposed transformation in Section 6.2 we use the disjuncts feature quite often, so we decided to copy the original classes into the generated metamodel, instead of referring to them.

6.1.2 The Additional Classes

The additional classes are two classes that are not based on the original metamodel. Those classes are used to structure the instances of the generated metamodel and to achieve the modularity we want. We already saw the additional classes in Section 5.1: Library and Function. Library will remain unchanged for our generated metamodels. Function does however has a reference to Tree. Since we can no longer assume that Tree is part of our domain, we can not keep this reference. Now we could make a version of Function for each class in the original metamodel. This would make our generated metamodel a lot bigger, so we decided to let Function have a reference to EObject instead of to Tree. Then during the model transformation we cast the EObjects to the required type.

6.1.3 The FunctionApply Classes

The FunctionApply class is quite similar to the FunctionApply of Figure 5.1. We have a FunctionApply class, that is a subtype of one of the original classes, takes a reference to another class and refers to a function using the function name. For an arbitrary metamodel we have often references to multiple different classes. With our extension we want to make it possible to use functions were we normally would use the original classes. So for each class that is used in a reference we need a FunctionApply that can be used for that class. We face this problem by collecting from the original metamodel all the classes for which there is a reference going to them. Then we generate for each of these classes a FunctionApply class. Each FunctionApply class gets a supertype relation to the original class.

Next we need a way to let the FunctionApply elements refer to the argument. This argument can be an element of any of the classes. We cover this in the same way as we did for Function class: the argument reference is of type EObject, allowing any class of model elements to be given and let the model transformation handle the problem of casting it to the right type.

In addition we also add to our FunctionApply classes a reference to the Function class and we give them an annotation to indicate that they are a FunctionApply class, such that they can be recognised as a function application during the model transformation.

6.1.4 The Param Classes

Similar to the FunctionApply elements, it must also be possible for the Param elements to be placed in our models when we require a reference. So to generate the Param classes we use the same approach as for the FunctionApply classes. For each class that has a reference to it in the original metamodel we create a Param class, such that each Param class has a supertype relation to the class of the original metamodel. To complete the Param classes we add an additional supertype relation to the original class and we add an annotation to indicate that they are Param classes.

6.2 Generating the Model Transformation

Now that we can generate the extended metamodel containing the function classes for any arbitrary metamodel, we also need to generate the model transformation to transform instances of the extended metamodel to instances of the original metamodel. Those model transformations are based on the model transformation as shown in Section 5.2. To generate those we take the extended metamodel and generate the model transformation. The file containing the model transformation is a QVT Operational file, containing text. So a model-to-text transformation is required for this.

There are three main model-to-text tools for EMF: JET, Acceleo and Xpand. We decided to use Acceleo, because from those three tools Acceleo is the most active one. The documentation for Acceleo also seems to be more complete and easier to access compared to the other two tools. A disadvantage of using Acceleo, is that Acceleo only accepts one model as input model and we need both the original metamodel and the extended metamodel to generate the model transformation. To overcome this we create a small container model, which contains references to both the original metamodel and the extended metamodel.

For our model transformation we have to generate mappings such that any element from the source model (an instance of the extended metamodel) is mapped to the target model (an instance of the original metamodel). Our generated model transformation will become similar to our model transformation described in Section 5.2, but now we want to generate the model transformation for any given metamodel as the original metamodel. We will start with the original classes.

The complete Acceleo transformation can be found in Appendix B.

6.2.1 The Original Classes

The original classes will again be transformed in a trivial manner: all the attributes and the references of the element in the source model are added to the element in the target model.

6.2.2 The FunctionApply Classes

The following step is the transformation of FunctionApply elements. When we defined the mapping for the FunctionApply elements in Section 5.2, we used a parameter in the mappings that takes the role of the parameter for when we are applying a function. For example when we have a FunctionApply element in our model, then it needs a reference to an element. Now when we were transforming the FunctionApply element, we used the element we refered to as the parameter for the transformation of the function we want to apply. So we execute the mapping of the function with as parameter this element. In the transformation for the Tree metamodel, the type of the parameter would always be Tree. Now we want to generate the transformation for any given metamodel, so it must be possible to use other types for the parameters.

To solve this we decide to allow any type for the parameters. Then when we have to transform we look at the type of the parameter and transform the parameter using the transformation belonging to the type. We implemented this transformation using a query in our model transformation. This query looks at the type of the element and then transform the element using the mapping that handles that type. Now that we can transform the parameter to any type we can easily implement the mapping for the FunctionApply classes.

6.2.3 The Param Classes

Our mapping for the Param classes is even more straight forward. For this we take the same approach as in Section 5.2. But instead of using the Tree type, we cast the parameter to the type the Param class belongs to.

6.2.4 The Complete Model

Now that we can transform all the elements of the source model to the target model, we can take the root of the source model and transform this to the root for our target model. In order to do so we use the query which we also used for the FunctionApply classes. This query looks at the type of the root element to execute the required mapping.

6.3 Streamlining the Procedure

We now have two steps in our procedure: one to generate the extended metamodel and one to generate the model transformation. Those steps are implemented in separated Eclipse projects and to execute the transformations they have to be called manually. Executing the transformation is now quite cumbersome, because you have to find the transformation files in the projects and you have to configure them to use your metamodel as the original metamodel. We want to streamline this procedure such that anyone can use it without requiring knowledge about the project and the transformation tools.

One way to combine the transformations would be to create a script that executes them after each other. QVT Operational has built-in support for ANT, where they generate an ANT script for you which executes multiple transformations after each other. However Acceleo has no such thing for ANT and calling the java code from an ANT script gives problems when you are calling it from

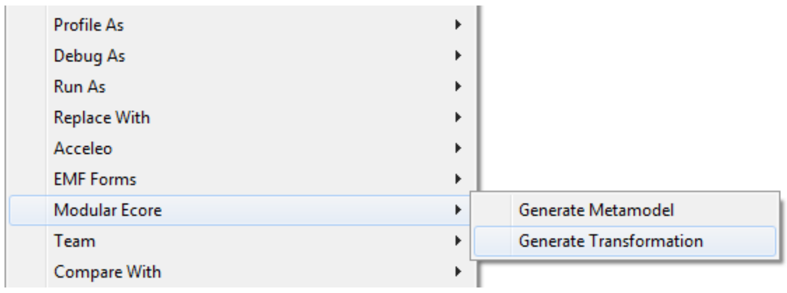

Figure 6.1: The commands added for ecore files

outside the plug-in environment.
When we try to execute the transformations from Java code the same problem arised. Using the tools from a stand-alone environment gives problems with the dependencies which are only satisfied inside the plug-in environment.

Because of those difficulties we decided to create an Eclipse plugin that will use both transformations. Creating a plug-in also gives the advantage that we can add extensions to Eclipse. Using the extensions we can add options to all kinds of menus to execute the transformations. A disadvantage of creating a plug-in could be that it only works for one IDE. This is however not a problem for us, as we are extending EMF which is targetting the Eclipse IDE.

The plugin depends on two other plugins that contain the Acceleo transformation and the container model containing the two EPackages (the container model is briefly introduced in Section 6.2). The QVT Operational transformations are located in the main plugin. We add two commands to the popup menu that appears when you right-click an ecore file. One to generate the extended metamodel and one to generate the model transformation. The commands are added using the extensions and extension points system supplied by Eclipse. An example of how this popup menu appears in the IDE can be found in Figure 6.1.

The commands are handled by java code inside the main plugin. For the generation of the extended metamodel we take the metamodel found at the selected file and use QVT to execute the transformation which we introduced in Section 6.1. This gives us the extended metamodel which we save in the directory of the selected file.

The generation of the model transformation requires an additional step. First we have to generate the container model, which will contain both the original model and the extended metamodel. For this step we assume that the extended metamodel is located where it would be placed after executing the generation of the extended metamodel. If this is not the case we execute the command of generating the extended metamodel, such that it will be placed there. Now we generate the container model. The container model is generated with a QVT Operational transformation, so we handle this similar to the generation of the extended metamodel. Instead of saving this container model in the directory, we keep the container model in memory, such that we can use the container model for the next step without having to save it on the file system. The second step is to execute the Acceleo transformation which we introduced in Section 6.2. To call the Acceleo transformation we use the plugin containing the Acceleo transformation. This plugin also contains code that is generated from the Acceleo transformation. By calling this code we can generate the required model transformation. Finally we save this model transformation in the directory of the selected file. Now the user can use the generated model transformation to transform models of the extended metamodel to models of the original metamodel.

The source code for this plugin can be found on Github[1].

6.4 Workflow

Now that the procedure has been automated, the workflow can be executed with much less manual effort. To use the workflow, we assume that the Eclipse plugin has been installed and we start with a metamodel, which we use as the original metamodel.

1. Generate the extended metamodel, by using the command that has been added to the popup menu by the plugin.
2. Use EMF to generate and editor for the extended metamodel.
3. Generate the model transformation, for which there also has been a command added to the popup menu.

Now we can use the generated editor to create Library models. And when we have created a model using the functions of the library, we can use the transformation to transform the model to a model instance of the original metamodel.

[1]https://github.com/StefvanSchuylenburg/functional-ecore

Chapter 7

Use Case: POOSL

POOSL[1] is a project that offers a method to describe and simulate system architectures. POOSL is mostly used for embedded systems. It offers the POOSL language in which you can describe processes. For this use case we want to extend the POOSL language with modularity over the elements that are found in the language. There are already some ways to achieve modularity in POOSL by using the *method*, *system* and *import* elements of the language. However, with our approach we aim to support modularity over the instances of the language in a more general way. We want to allow users to reuse elements of the language which they could not reuse before. With this addition we hope to make the language more pleasant to work with.

The POOSL language is implemented using Xtext[2]. In our approach we have not targeted Xtext yet, so for this use case we will extend the Xtext grammar manually. To extend the POOSL language with modularity we first take the metamodel of the POOSL and we generate the extended metamodel for it. Then we extend the Xtext grammar. In the next section we will look at the classes that we have added to the metamodel and how we adapted the language to use those classes.

7.1 Extending the Grammar

When extending the grammar, we could not give the grammar the same freedom as was possible in the metamodel however. In our solution the FunctionApply class has an argument which can be of any type. To apply this to the POOSL grammar is not directly possible, because it breaks the LL-parsing by allowing multiple language elements which the LL-parser can not tell apart. There are multiple ways to solve this. We could for example request a certain token before the argument such that the users can indicate what language element they are using. This is not feasible however, because the user of the language may not be aware of what language element they are using. Some language elements may look very similar to the user (for example Statement and Expression), but are different for the grammar. We worked around this by limiting our solution to only a few elements of the language for this use case. We only added support

[1]http://poosl.esi.nl/

[2]https://eclipse.org/Xtext/

for the Statement and Expression classes, as it was not possible to use those as a parameter in POOSL yet and for the users of the language it was desirable to use those elements in a more free way. The Statement and Expression classes are used to describe the actions a process can do and to manipulate the data. We also limited the FunctionApply classes, such that they only extend the Statement and Expression classes. Furthermore we limited the Statement FunctionApply elements to only accept Statement elements as arguments and the Expression FunctionApply elements to only accept Expression elements as arguments. In the Xtext grammar the Statement and Expression definition are modified to allow the FunctionApply class as an alternative.

Next we must also add the Function, Library and the Param classes to the grammar. For the Function class we encounter the same problem as for the FunctionApply classes, because the body of a Function can be of any type. However, this time it is not possible to see whether a Function element is used as a Statement or as an Expression, because the Function elements are all placed in the Library without additional context. So instead of deriving the class of the body from the context, we have to let the creator of the function to explicitly state whether he is defining a Statement function or an Expression function. We expect that this is not a problem for the developers of the functions, because developing those functions already require a basic knowledge of how the language is constructed and what elements you are using.

Next we have the Library class, which is simply a collection of functions. For our solution we intended to be able to create a stand-alone file containing the Library, just like we did for the models. To achieve this, it must be possible to use the Library as a root of the POOSL language extension. We achieved this by editing the metamodel, such that the root class of the POOSL extension also contains a reference to the Library class. Editing the metamodel for this has a disadvantage, because the metamodel is generated. So each time when we generate the metamodel again, we also have to edit the metamodel file again. A second disadvantage is that the root class is one of the original classes. So if we want to use a reference to the original metamodel instead of copying all the classes of the original metamodel, then this approach would not be possible.

The Param classes are implemented in the grammar in exactly the same way as the FunctionApply classes. For the Param classes we could again use the context, because you only use ParamStatement elements where Statement elements are required and you only use ParamExpression elements where Expression elements are required.

7.2 Future Work

This use case is still limited and requires a number of adaptations to make it work. A next step would be to automate the adaptations of the original grammar for the extended metamodel. For example it could be possible to generate the grammar definition for the new classes and to adapt the original grammar definitions to include the new classes. Automating this process would make it easier to update the grammar when changes have been made to the original grammar or to the metamodel.

Furthermore, it would be interesting to find a way to overcome the limitations of using the classes, such that we could indeed use any element as argument for the FunctionApply elements. Possible solutions for this are to let the creator of the Function elements define what the types of the parameters are or to create languages where all the elements of the language are recognised as different by the LL-parser.

Finally it would also be better if we could adapt the grammar without having to edit the original classes as we did for the Library class.

Chapter 8

Related Work

Other researches have already looked at how modularity can be introduced into EMF and to MDE in general. Kolovos et al. [3] have stressed the importance of modularity. In their paper they claim that the scalability of MDE is heavily limited; they claim that when models become large and complex, they become difficult to work with. They compare how domain-specific languages are constructed to how Java and the Java Development Tools are constructed and how modularity is achieved for Java. The authors show two aspects related to modularity in modeling, to show the mistakes that are often made by designers of domain-specific languages. The paper does however not give an explicit solution to the problem of achieving modularity in modeling.

Kelsen et al. tackle the modularity problem in their paper [4] by introducing a mathematical approach to decompose a model into submodels. In the paper the authors provide definitions for metamodels, models and submodels. Using those definitions they propose an algorithm to decompose a model into submodels. Based on the algorithm they provide a plugin for EMF that can decompose models created with the EMF framework.

Heidenreich et al. [5] propose two ways of extending modelling languages for modularity. The first involves extending the metamodel of the language to add new element types to the model, which act as interfaces between submodels. The second does not involve extending the metamodel, but does extract the interfaces implicitly. The interfaces are used as anchor and slot components in the models. Then when components have been created with the anchor and slot interfaces they can be linked together to create a composite model from the submodels. The interface system is comparable to our solution of using functions, but with our function system we intend to create a more reusable solution.

In another approach proposed by Kelsen et al. [6] modularity is achieved by creating submodels with interfaces. Those submodels can be composed by creating a model which contains referential nodes to the submodels. The interfaces are certain relations that are found in the metamodel, giving information about which relation the submodel will satisfy.

Another solution is the Common Variability Language (CVL) tool. CVL is a language that can be combined with a Domain-Specific Language (DSL) to create an environment in which you can use variables. CVL is introduced by Haugen et al. [7] to use variables in a DSL without affecting the original

DSL. The variables will be bound in a separate model. Using the binding of the variables a base model (defined in the given DSL) can be transformed to a new model. A CVL tool has been implemented. The implementation is based on EMF and supports the domain-specific languages that are defined through EMF.

Furthermore, there is a plugin for EMF that aims to achieve modularity in yet another different way. The plugin *EMF Splitter* [8] allows models to be split based on a modularity strategy. The models can be split into submodels, such that each submodel gives a view on one part of the model. The fragmentation is defined on the metamodel and when creating a new model the submodels will be placed into packages of an Eclipse project, with each package holding one view of the model. EMF Splitter also supports the composition of the submodels into a single model, such that the model can be used for the tooling that exists for the metamodel.

In this thesis we showed another solution. Our solution is based on referencing to model parts from other models, similar to the solutions of [5] and [6]. What our approach does different, is using functions.

Our solution is quite similar to [5], because they also introduce new model elements which work somewhat similar to our FunctionApply elements. Like our solution, you can put those elements in your model, which are used as a placeholder for the model part which is defined in a separate model file. The differences are that they define the linking of the model parts in a different model file and that there is no support for parameters. Advantages of their solution, compared to the solution presented in this paper, are that is easier to change how the model parts are linked and that model parts can be linked using multiple anchor points. For example, when you have a model part that has two different anchor points, then you can link another model part to it on two different places. Advantages of our solution are that parameters can be used and that no separate model to link the model parts together is required. Using the parameters, you can use more reusability, because the model parts are usable for more situations.

Our solution is also quite similar to CVL, as introduced in [6]. With CVL you can also reuse model parts and change properties of the model parts while using them. The differences are that CVL uses a separate model file which changes the values of attributes and model elements and that CVL works with model instances of the original metamodel. Using model instances of the original metamodel, gives the advantage that you can use the tooling that is created for the original metamodel. Furthermore, CVL has a more extensive plugin, which has already been used by a number of companies. Advantages of our solution are that no separate model is needed to combine the model parts and that functions are used within models directly. Using functions within models directly, has the advantage that it follows the convention as seen in programming languages such as Java and C, where you have a main function that contains calls to other functions. Using those conventions, can make it easier for software developers to start using our solution, because they are already familiar with it.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this paper we looked at how modularity can be improved when working with models in EMF. We presented a solution based on extending existing metamodels. The extended metamodels contain new concepts. Those concepts allow developers to define and use functions. We also presented a transformation to transform models with functions to models without functions. Those steps are then automated, such that the solution can be applied to any metamodel. The system that automates the steps can be installed in Eclipse as a plugin.

With our new solution, it is possible to separate one model into multiple model parts and to reuse the model parts in a parameterized way. The extended metamodel enhances the domain-specific language such that libraries of functions can be created in separate models. Those models can then be used in other models to keep those models concise and better maintainable. We expect that our solution helps developers to work with models in a more modular approach, without having to depend on whether the original metamodel supports modularity.

9.2 Research Goals Revisited

In Section 1.1 we introduced four research goals. In this section we will argue for each research goal how we achieved it.

Our solution has also been implemented. The implementation can be found at `https://github.com/StefvanSchuylenburg/functional-ecore`. One could argue, however, that the implementation is not yet complete to be used for production. Suggestions for how our solution can become complete can be found in Section 9.3. The implementation does serve as a proof of concept; it shows that it can be implemented and gives a basis on which future improvements can be build.

9.2.1 Changeability

We defined changeability as the ability to change one part of the model without affecting the other parts. Without the usage of functions it is already possible to keep parts of the model separated. This can be done for example by structuring the model such there are as few as possible relations between parts of a model. One part can then be edited without affecting the other parts of the model. Our solution improves this by making clear what the parts are that can be changed without affecting the other parts.

Without functions, it may not be clear what the parts are that should stay loosely connected, because all the parts are bundled together in a single model file. With functions it is clear what the separated parts are, as they are physically separated. The parts are placed under different functions and are given names.

So, to improve changeability our solution does not introduce new things that were not possible before. It does, however, help in making clear how the models are structured, such that better changeability can be achieved.

9.2.2 Comprehensibility

Comprehensibility is the ability to understand a part of the model without having to understand the complete model first. As the models become larger and there are more model elements and relationships, it becomes more difficult to keep the model comprehensible. With our solution, this can be solved by keeping one model as the main model. This is the model that will be transformed to the model we want to use. From this model you can refer to other functions. Now when a developer looks at the model, he sees the names of the functions and he finds less elements than the complete model contains. This gives him information about how the element are related in the model, without having to look at all the elements that are part of the complete model.

For example with the functions introduced for the Tree model in Chapter 5, we could make one tree consisting of a Node element with two different FunctionApply elements as children. When you look at the model now, you can see that the children are separated parts. And using the name of the Function you can get an idea what the parts do, without having to know how they are precisely constructed.

9.2.3 Reusability

Reusability is the ability to reuse parts of models in multiple different models. EMF itself does not supply reusability of model parts.

Our function system achieves this by allowing developers to create libraries of functions. Then in other models you can use the functions of the libraries by referring to them. This gives a high level of reusability, because the libraries can be placed in separate files.

9.3 Future Work

Our solution can, however, still be improved. For example we do not support using attributes as parameter; with the current solution, you can only use the parameters for references of model elements and not for the attributes of the model elements.

Support for basic constructs like loops and if-then-else structures are also not possible yet. In general-purpose languages those constructs are often used to create more powerful functions. Adding this to model functions could greatly improve the flexibility of functions. Using those constructs it would also be necessary to create expressions over arguments, such that we can evaluate the guards of the loops and evaluate the if-then-else conditions.

Another feature that is still missing in our solution is validation of how the functions are used within models. Right now you can use any type of element as argument for a function application. Validation of whether the type of the argument satisfies the type of the function application could help developers to discover mistakes early. Especially because the Ecore models are based on strict typing, it can become confusing for developers when we do not handle type checking. This also applies to checking whether a Function Apply element has the same expected type as the body of the Function element.

9.4 Acknowledgments

This work was supported in part by the European Union's ARTEMIS Joint Undertaking for CRYSTAL - Critical System Engineering Acceleration - under grant agreement No. 332830.

Bibliography

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] Object Management Group, "Omg formally released versions of mof." `http://www.omg.org/spec/MOF/`.

[3] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Scalability: The holy grail of model driven engineering," in *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering*, pp. 10–14, 2008.

[4] P. Kelsen, Q. Ma, and C. Glodt, "Models within models: Taming model complexity using the sub-model lattice," in *Fundamental Approaches to Software Engineering*, pp. 171–185, Springer, 2011.

[5] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler, "On language-independent model modularisation.," *T. Aspect-Oriented Software Development VI*, vol. 6, pp. 39–82, 2009.

[6] P. Kelsen and Q. Ma, "A modular model composition technique," in *Fundamental Approaches to Software Engineering*, pp. 173–187, Springer, 2010.

[7] Ø. Haugen, B. Moller-Pedersen, J. Oldev, G. K. Olse, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Software Product Line Conference, 2008. SPLC'08. 12th International*, pp. 139–148, IEEE, 2008.

[8] A. Garmendia, E. Guerra, D. S. Kolovos, and J. de Lara, "Emf splitter: A structured approach to emf modularity," in *XM 2014–Extreme Modeling Workshop*, p. 22, 2014.

Appendix A

Transformation: Extending the Metamodel

This appendix contains the model transformation that is used in Section 6.1. As inputs it takes a metamodel and it generates a new metamodel. Developers can use the generated metamodel to create model instance, in which they can create and use functions.

```
modeltype Ecore uses 'http://www.eclipse.org/emf/2002/Ecore';

/**
 * The transformation takes a normal ecore model
 * and add function classes to it.
 */
transformation addFunctions(in _source: Ecore, in _ecore: Ecore,
  out _target: Ecore);

main() {
  // map for the target model
  var root := _source.rootObjects()![EPackage];
  root.map transform();
}


/**
 * Transform the EPackage (that is the package holding all the classifiers)
 * of the source to the package of the target.
 */
mapping EPackage::transform(): EPackage {
  name := self.name + "_fun";
  nsPrefix := self.nsPrefix + "_fun";
  nsURI := self.nsURI + "/functional";

  // Creating the model elements
  var function := object EClass {
    name := "Function";
    eStructuralFeatures := Set{
      object EReference {
        name := "body";
        containment := true;
        lowerBound := 1;
        upperBound := 1;
        eType := ecore("EObject");
      },
      object EAttribute {
        name := "name";
        eType := ecore("EString");
      }
    };
  };
  var lib := object EClass {
    name := "Library";
    eStructuralFeatures := object EReference {
      name := "functions";
      containment := true;
      lowerBound := 0;
      upperBound := -1;
      eType := function;
    };
  };

  var functionClasses := self.referencedClasses()->map
    toFunctionClass(function, self.nsURI + "/functional");
  var paramClasses := self.referencedClasses()->map
    toParamClass(self.nsURI + "/functional");
  eClassifiers := self.eClassifiers
    ->union(functionClasses)
    ->union(paramClasses)
    ->union(Set{lib, function});
}

/**
 * Creates a function class that is a subtype of self.
 * The given function class, is the class defining a function.
 * @param nsURI the name space of the target model
 */
mapping EClass::toFunctionClass(functionClass: EClass,
    nsURI: EString): EClass {
  name := "FunctionApply" + self.name;
  eSuperTypes := self;

  eAnnotations := object EAnnotation {
    source := nsURI + "/FunctionApply";
  };

  eStructuralFeatures := Set{
    object EReference {
      name := "function";
      eType := functionClass;
      containment := false;
      lowerBound := 1;
      upperBound := 1;
    },
    object EReference {
      name := "argument";
      eType := ecore("EObject");
      containment := true;
      lowerBound := 0;
      upperBound := -1;
    }
  };
}

/**
 * Creates a Lambda class htat is a subtype of self
 * @param nsURI the name space of the target model
 */
mapping EClass::toParamClass(nsURI: EString): EClass {
  name := "Param" + self.name;
  eSuperTypes := self;

  eAnnotations := object EAnnotation {
    source := nsURI + "/Param";
  };

  eStructuralFeatures := Set {
    object EAttribute {
      name := "index";
      eType := ecore("EInt");
      defaultValueLiteral := "1";
    }
  };
}


/**
 * Gets all the classes in the EPackage with a reference to it
 */
query EPackage::referencedClasses(): Collection(EClass) {
  var refs := self.eClassifiers.allSubobjectsOfKind(EReference)
    .oclAsType(EReference);
  return refs.eReferenceType->selectByKind(EClass);
}

/**
 * Finds the Classifier with the given name in the Ecore model
 */
query ecore(name: EString): EClassifier {
  return _ecore.objectsOfType(EClassifier)
    ->any(classifier: EClassifier | classifier.name = name);
}
```

Appendix B

Transformation: Generating the Model Transformation

This appendix contains the model transformation that used in Section 6.2. It takes as input a container model, which contains a reference to an original metamodel and an extended metamodel. The model transformation generates a text file, which is a QVT Operational transformation. The generated transformation transforms a model instance of the extended metamodel to a model instance of the original metamodel. The extended metamodel that is used for this transformation, should be generated by the transformation of Appendix A.

```
[comment encoding = UTF-8 /]
[module generate(
  'http://stefvanschuylenburg/functionalecore/packagecontainer',
  'http://www.eclipse.org/emf/2002/Ecore'
)]


[template public generateTransformation(container : EPackageContainer)]
[comment @main/]

[let nsURI: EString = container.extension.nsURI]
[file (container.extension.name.concat('.qvto'), false)]
modeltype M uses '[container.original.nsURI/]';
modeltype MFun uses '[container.extension.nsURI/]';

transformation [container.extension.name/](in source: MFun, out target: M);

main() {
  var root := source.rootObjects();

  // map the root based on its type
  root.transformAny(null);
}

/*
```

```
 * Mappings for the referenced classes.
 * Disjuncts between all possible instances.
 */
[for (class: EClass | container.extension.referencedClasses(nsURI))]
mapping MFun::[class.name/]::transform(in x: Sequence(Element)):
    M::[class.name/]
  [let subclasses: OrderedSet(EClass) =
    container.extension.subClasses(class)->asOrderedSet()]
  disjuncts MFun::[subclasses->first().name/]::transform
    [subclasses->first().name/][for (subclass: EClass |
      subclasses->excluding(subclasses->first()))],
    MFun::[subclass.name/]::transform[subclass.name/][/for];
  [/let]
[/for]

/*
 * Mappings for objects from [container.original.nsURI/]
 */
[for (class: EClass | container.original.eClasses())]
mapping MFun::[class.name/]::transform[class.name/](in x: Sequence(Element)
    M::[class.name/] {
  [if (not class.eAllAttributes->isEmpty())]

  // the attributes
  [for (attr: EAttribute | class.eAllAttributes)]
  [attr.name/] := self.[attr.name/];
  [/for]
  [/if]
  [if (not class.eAllReferences->isEmpty())]

  // the references
  [for (ref: EReference | class.eAllReferences)]
  [ref.name/] := self.[ref.name/].map transform(x);
  [/for]
  [/if]
}
[/for]

/**
 * Mappings for the FunctionApply and Param elements
 */
[for (class: EClass | container.extension.referencedClasses(nsURI))]
mapping MFun::FunctionApply[class.name/]
  ::transformFunctionApply[class.name/](
    in x: Sequence(Element)): M::[class.name/] {
  init {
    var function :=
      self.function.body.oclAsType(MFun::[class.name/]);
    var argument :=
      self.argument.oclAsType(Element).transformAny(x);
    result := function.map transform(argument);
  }
  end {
    // remove argument again: it is only passed around
    argument->forEach(arg) {
      target.removeElement(arg);
    };
  }
}
mapping MFun::Param[class.name/]::transformParam[class.name/](
    in x: Sequence(Element)): M::[class.name/]
  when {x != null && self.index > 0} {
  init{
    result := x->at(self.index).deepclone()
      .oclAsType(M::[class.name/]);
  }
}

[/for]


/**
 * Transforms an element using the mappings defined in this transformation.
 * The transform is based on the type of the element
 */
query Element::transformAny(in x: Sequence(Element)): Element {
  switch {
    [for (class: EClass |
      container.extension.referencedClasses(nsURI))]
    case (self.oclIsKindOf(MFun::[class.name/])) {
      return self.oclAsType(MFun::[class.name/]).map
        transform(null);
    }
    [/for]
  }
}

[/file]
[/let]
[/template]


[comment
  All the concrete EClasses within ePackage,
  this will not yield the abstract classes
/]
[query public eClasses(ePackage: EPackage): OrderedSet(EClass) =
  ePackage.eClassifiers->selectByKind(EClass)->reject(abstract)
/]
```

```
[comment
  Selects all the concrete classes that are a subtype of superClass
/]
[query public subClasses(ePackage: EPackage, superClass: EClass):
    OrderedSet(EClass) =
  ePackage.eClasses()->select(class: EClass |
    superClass.isSuperTypeOf(class))
/]

[comment
  Finds the referenced classes by looking at for which classes there
  is an FunctionApply method.
/]
[query public referencedClasses(extension: EPackage, nsURI: EString):
    OrderedSet(EClass) =
  extension.eClassifiers->selectByKind(EClass)
    ->reject(class: EClass | class.isFunctionApply(nsURI))
    ->select(class: EClass | extension.eClasses()
      ->exists(class2: EClass | class.isSuperTypeOf(class2)
        and class2.isFunctionApply(nsURI)))
/]


[comment Determines whether the class is a FunctionApply class. /]
[query public isFunctionApply(elem: EModelElement, nsURI: EString): Boolean
  not (elem.getEAnnotation(nsURI + '/FunctionApply') = null)
/]
```