Instance Based Meta-Model Generation

Simon Van Laerhoven

Abstract

Model-driven engineering aims to improve the efficiency of the software development process. Domain-specific modelling languages (DSMLs) are used to develop models and are specified through meta-models. Domain experts, who have the knowledge about the domain usually don't have the skills needed to develop a correct meta-model, while software engineers, who do have the needed skills, don't have enough knowledge about the specific domain. This makes close interaction between software engineers and domain experts necessary which slows the development process. This is where the need for instance based meta-model generation rises. In this paper we propose a way to create sample models using a generic meta-model in AToMPM. With model transformations these samples will be used to automatically generate a meta-model to which all the samples conform.

Keywords: Meta-Model, Model, Example-driven modelling, Exploratory modelling

1. Introduction

Model-driven engineering (MDE) has gained a lot of populairty over the years. It is a higher level form of software development which allows domain experts, with very limited knowledge of programming, to create software applications. Before a domain expert can start modelling, a domain specific modelling language (DSML) has to be defined to match the problem domain and constrain the user. A DSML can be defined using a meta-model (MM). Domain experts, who have the knowledge about the domain usually don't have the skills needed to develop a correct meta-model, while software engineers, who do have the needed skills, don't have enough knowledge about the specific domain. This makes close interaction between software engineers and domain experts necessary which slows the development process. Instance based meta-model generation is a possible solution for this problem. The idea

is that with multiple sample models, created by the domain expert, a meta-model is automatically generated. All the sample models will conform to the created meta-model. With this the need for close interaction with the software engineer decreases drastically, although the software engineer might still be usefull for some of the steps in the meta-model generation process as we will discuss later.
In this paper we will propose a method for meta-model generation in AToMPM (A Tool for Multi-Paradigm Modelling) (Syriani et al., 2013).

2. Related Work

The following papers are somewhat related to the problem of Instance based meta-model generation. But they don't provide much help except that they give context to the problem at hand.

2.1. Generating meta-model-based freehand editors

Minas (2007) talks about freehand editing of meta-model-based models as opposed to structured editing. Most tools only provide structured editing of models, where all the allowed operations transform a correct model into another correct model. Freehand editing allows editing of a model without any restrictions, giving more freedom to the user. They combine meta-model-based models with freehand editing in the tool DiaMeta.

2.2. Example-driven meta-model development

Example-driven meta-model development by López-Fernández et al. (2015) is the most relevant paper in our case. They propose an approach where a meta-model is generated from model fragments which are specified using the tools Dia or yED. The meta-model is iteratively updated using the model fragments and refactoring by the user. A virtual assistant proposes some refactorings to the user during the process. In the end the meta-model is validated by the domain expert by again checking example models against the meta-model.

2.3. Automated Model-to-Metamodel Transformations Based on the Concepts of Deep Instantiation

In Kainz et al. (2011) they propose an automated model to meta-model transformation in the Eclipse Modelling Framework (EMF). The tranformation is divided in phases and based on the concept of deep instantiation.

Deep instantiation allows elements in a model to be both a class and an objects (clabjects). The result of the different processing phases will be a complete definition of the meta-model.

2.4. Supporting constructive and exploratory modes of modeling in multi-level ontologies

Atkinson et al. (2011) talks about the differences between constructive and exploratory modelling. Constructive modelling aims to create a complete, definitive description of all the types in a system. Building a meta-model can be seen as constructive modelling as it completely describes all types system. Exploratory modelling, on the other hand, aims to develop the types that characterize the objects in a domain. Instance Based Meta-Model Generation can then be seen as a form of exploratory modelling.

3. Model to Meta-Model Transformations in AToMPM

In this section we will propose a method to transform a collection of sample models into a meta-model which all sample models conform to. The tool we work with is AToMPM (Syriani et al., 2013). It allows us to create meta-models, models and model tranformations with a visual syntax. The process will be executed in 7 steps.

3.1. Generic Meta-Model

First we need a generic meta model which we will use to model the example models with. Figure 1 represents the abstract syntac of our generic meta-model. There are two classes: Model and Object. A Model object represents one sample model and contains zero or more Objects. An Object has as attributes *type* and *attributes*. The attributes given to an Object can be of the following types:

- boolean
- int
- float
- string
- list

- map

The items in *list* and *map* can be any of the mentioned types. There can exist associations between Objects. These can have attributes and a type as well. If no type is given to an association from object a of type A and object b of type B, the type of the type of the association will be "AtoB". Figure 2 shows how we will represent our generic meta-model.

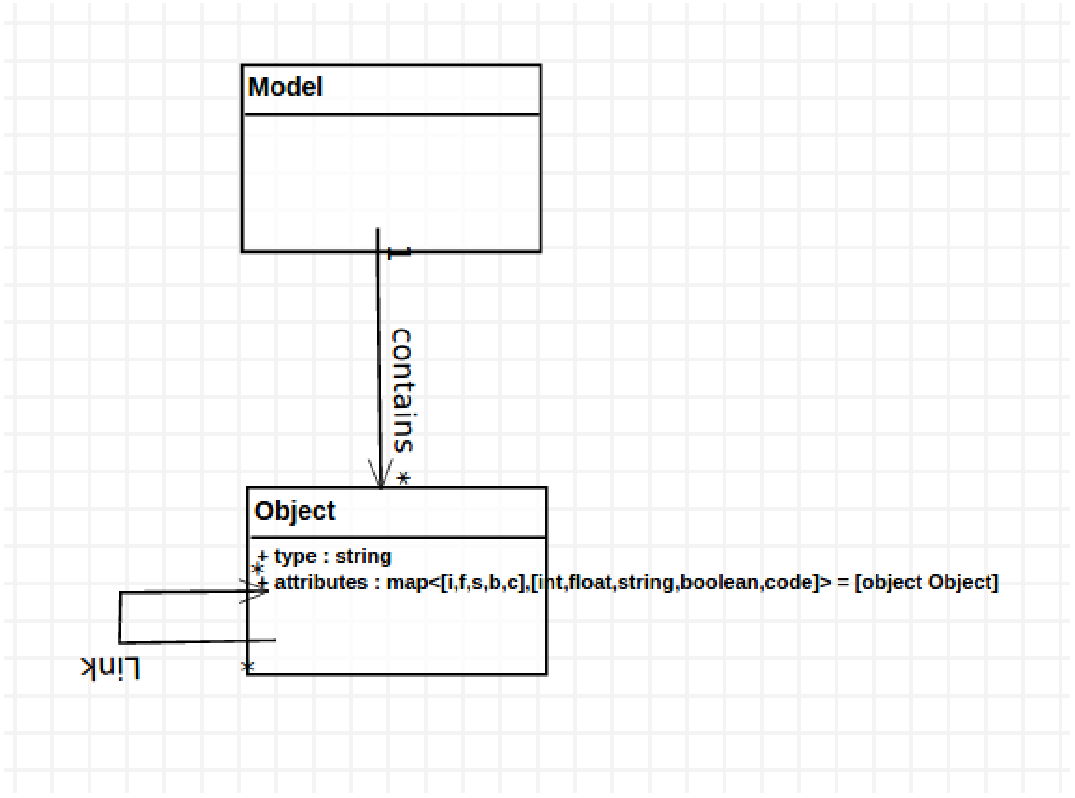
Model
1
contains
*
Object
+ type : string
+ attributes : map<[i,f,s,b,c],[int,float,string,boolean,code]> = [object Object]
*
*
Link

Figure 1: Generic Abstract Syntax

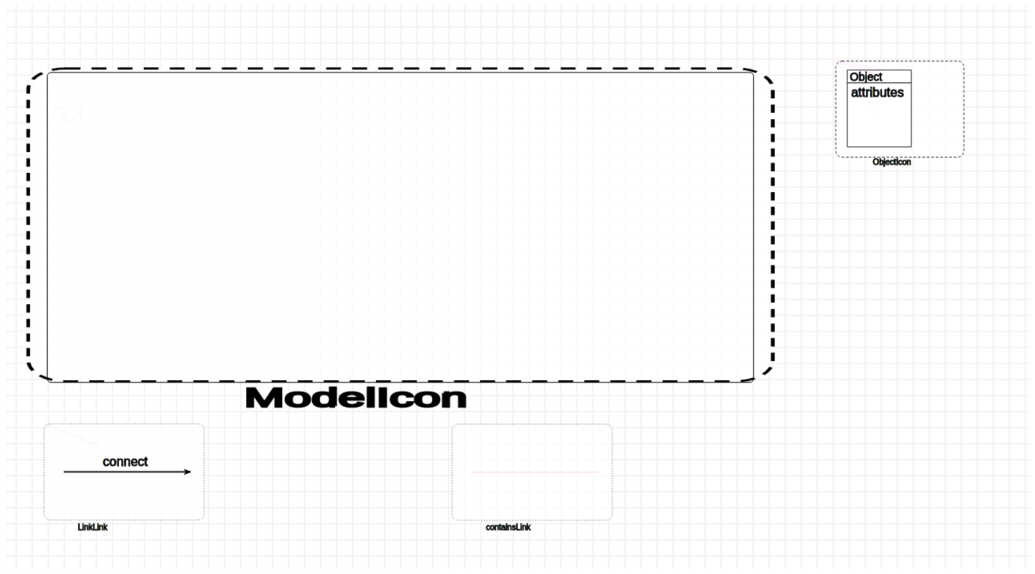

Figure 2: Generic Concrete Syntax

3.2. Create Example Models

With our new generic meta-model a domain expert can create a number of sample models for the domain they desire. These models should represent the most important, if not all, use cases of the desired DSML. Some samples of a DSML representing a railway system can be seen in Figure 3. Note that there are many more than two samples needed to create a decent MM for our railway system. When we have a model file containing samples we run the script *createFiles.sh* with as first argument the model file and as second argument the name of the DSML that we wish to create. In the folder GeneratedMMs/<chosen name>four files are generated. An abstract syntax file, a concrete syntax file, a compiled metamodel file and a compiled concrete syntax file. The abstract and concrete syntax files are just copies of the sample model which have to be transformed using the generateAS and generateCS transformations respectively. The compiled files are empty files which can be selected when compiling the model files.

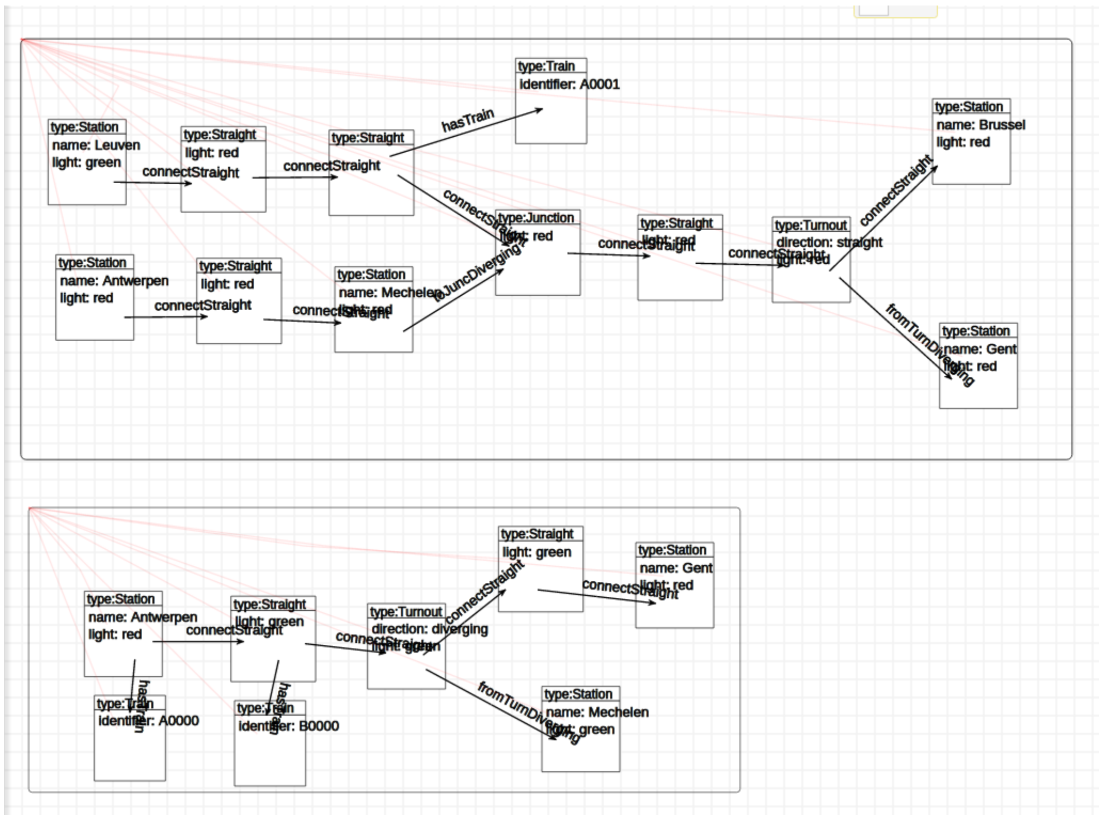

Figure 3: Two samples of a DSML representing a railway system

3.3. Model transformations into abstract syntax

Through the AToMPM model transformations we are able to automatically generate a meta-model describing the language of the sample models. This transformation should be done in another than the original sample model file as we lose the original model. In case *createFiles.sh* was used select the <chosen name>MM.model file for this transformation. The general workflow is described in the following steps.

1. **Create Classes** For every object, a new class object is created with as name the type of the corresponding object. For each attribute the object has, the type of that attribute is determined and converted to a javascript type.
2. **Merge Classes** All class objects that have the same name are merged together. All attributes from both class objects we be copied into the

merged class object. If two attributes have the same name, but a different type, then a choice is made to use the type that can hold the most information, similar to the approach in López-Fernández et al. (2015). Our types are ordered in the following ordering: map >list >string >float >int >boolean. Every higher ranked type can hold the information of all lower ranked types and more. e.g. a map of integers to integers can represent a list of integers or a string containing the value "42" can represent an integer of value 42.

3. **Create Associations** For every association links between two objects a of type A and b of type B, an association link from class object A to class object B is created. If there is an association link between two objects of the same type or an object has an association link to itsself, the class object representing that type will have an association link to itsself as well. The created association link will have all the attributes the link between the objects had.
4. **Count Cardinalities** For every class we count the minimum and maximum incoming and outgoing association links of the objects of that class. These are set as the cardinalities of that class. e.g. in our railway system example an object of type *Station* has at most one outgoing link *hasTrain* to an object of type *Train*. So the class object *Station* can have zero or one outgoing associations of type *hasTrain*. It can happen that the maximum associations of a class should be unbounded, but this cannot be represented in the samples we induce our MM from. The user can change this later in the MM.
5. **Merge Associations** All association links with the same name, between the same class objects are merged together. The merging of the attributes of each association link is done in the same way as in the Merge Classes step.
6. **(optional) Create Super Classes** There is an option to automatically create abstract super classes. The abstract classes will have almost no impact on the DSML we are trying to create, but can make the MM much more readable. This is optional because it can do the opposite as well and make it less readable in some cases. A super class is created when two different classes A and B have the same incoming association from C (López-Fernández et al., 2015). The abstract super class will be named AB, the association from C will go to AB and A and B inherit from AB. Attributes with the same name in A and B will be pushed up to AB. This is where slight differences can occur when

using autmatically generated super classes. If the attributes have the same name, but different types, a choice between types is made.
A super class of two super classes can be made in the same way as described above. All superclasses that hold no value are discarded. e.g. when class A inherits from AB, which inherits from ABC, and AB has no attributes or associations, then A will directly inherit from ABC and AB will be discarded.

7. **Delete original sample models** The original sample models are removed from this file so the meta-model can be edited and interpreted more easily.

3.4. Manually check abstract syntax

At this stage the meta-model could be checked to see if it looks correct. This can be done by a software engineer checking the meta-model. For this again a short interaction between a software engineer and domain expert is needed. This step is optional because later in the process the meta-model can be checked by creating new models, which the domain expert considers correct, and trying to verify if they conform to the meta-model.

3.5. Default concrete syntax generation

From our newly generated abstract syntax, we will generate a default concrete syntax. A rectangle with a classname to represent an object of that class and an arrow to connect two objects representing an association link.

3.6. Manually change concrete syntax icons

The domain expert then needs to change the concrete symbols to the symbols that they had in mind.

3.7. Create models using newly generated MM

The final step is using the newly generated meta-model to create models. These models should be used to verify the meta-model. If a created model is correct according to the domain expert but is not accepted as a valid instance of the meta-model, then the meta-model needs to be adapted. When an incorrect model is accepted as a valid instance, the MM should be more restricted. This can't be done in the sample models as it does not allow negative models.

4. Iteration

When we have a MM, but we are not satisfied with the result, we can iteratively improve our MM. There are two ways of doing this. One option is to add more samples to our original model samples and regenerate the entire MM. If we did not manually improve our MM this is not a problem. However, if we did manually change our MM, then we would lose this progress. So another option is to add new samples to our existing abstract syntax. Again with model transformations we can transform this abstract syntax together with the new sample(s) into an updated version of the abstract syntax of the MM. This way our manual work with the MM is preserved. The update transformation is very similar to the model transformations mentioned earlier.

5. Usage

The following steps explain the usage of our meta-model induction transformations in AToMPM.

1. Import *MMGeneration.defaultIcons.metamodel* as a toolbar, and use this to create Models, and objects in these Models. The objects and association links between objects can be edited to set the type and attributes. Afterwards save this model.
2. In a terminal in the Formalisms/MMGeneration folder run ”createFiles.sh <path/to/sample/models> <MMname>” to create a working folder for our MM.
3. Open *GeneratedMMs/<MMname>/<MMname>MM.model* and apply *GenerateMM/generateAS/T_generateAS.model* to transform this file into the abstract syntax of our MM. Apply *GenerateMM/generateAS/T_generateASWithSuperClasses.model* if you wish to automatically create abstract super classes. Save this model.
4. Compile current model into an abstract syntax meta-model.
5. Open *GeneratedMMs/<MMname>/<MMname>MM.defaultIcons.model* and apply *GenerateMM/generateCS/T_generateCSfromSamples.model.* Save this model.
6. Compile current model into an icon definition model.
7. Now you can import the *GeneratedMMs/<MMname>/<MMname>MM.defaultIcons.metamodel* as a toolbar and use this to verify the meta-model.

8. If you wish to update the abstract syntax with new sample models, you open the abstract syntax model file. Then you draw sample models with the MMGeneration toolbar. Update the abstract syntax using the *GenerateMM/generateAS/UpdateMM/T_update.model* transformation.

6. Conclusion

We have stated the need for Instance Based Meta-Model Generation. We aim to further improve the efficiency of the development process by decreasing the work for software engineers. We presented some related work to put our work into context. Most of our work is similar to the MM-induction phase in (López-Fernández et al., 2015). We have presented how we transform samples of a modelling language into an abstract and concrete meta-model syntax in AToMPM. We have also shown how to iteratively improve our meta-model with new sample models.

7. References

Atkinson, C., Kennel, B., Goß, B., 2011. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In: Procs. 7th Int. Workshop on Semantic Web Enabled Software Engineering, Bonn (October 24, 2011).

Kainz, G., Buckl, C., Knoll, A., 2011. Automated Model-to-Metamodel Transformations Based on the Concepts of Deep Instantiation. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 17–31.
URL `http://dx.doi.org/10.1007/978-3-642-24485-8`$_3$

López-Fernández, J. J., Cuadrado, J. S., Guerra, E., de Lara, J., 2015. Example-driven meta-model development. Software & Systems Modeling 14 (4), 1323–1347.
URL `http://dx.doi.org/10.1007/s10270-013-0392-y`

Minas, M., 2007. Generating meta-model-based freehand editors. Electronic Communications of the EASST 1.

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H., 2013. Atompm: A web-based modeling environment. In: Demos/Posters/StudentResearch@ MoDELS. Citeseer, pp. 21–25.