Co-Creation of Models and Metamodels for Enterprise Architecture Projects

Paola Gómez
Universidad de los Andes
Department of Systems and Computing Engineering
Bogotá, Colombia
pa.gomez398@uniandes.edu.co

Mario Sánchez
Universidad de los Andes
Department of Systems and Computing Engineering
Bogotá, Colombia
mar-san1@uniandes.edu.co

Hector Florez
Universidad de los Andes
Department of Systems and Computing Engineering
Bogotá, Colombia
ha.florez39@uniandes.edu.co

Jorge Villalobos
Universidad de los Andes
Department of Systems and Computing Engineering
Bogotá, Colombia
jvillalo@uniandes.edu.co

ABSTRACT

The *linguistic* conformance and the *ontological* conformance between models and metamodels are two different aspects that are frequently mixed. Particularly, this situation occurs in the EMF framework and it has resulted in some well known problems. The most relevant to us is the incapability to load metamodels at runtime, or even to modify the metamodels already in use. In this paper we present a strategy to solve this problem by separating the ontological and the linguistic aspects of a metamodel and a metamodeling framework. The strategy has been implemented in a graphical editor and is motivated in the context of Enterprise Architecture Projects.

Keywords

MDE, EMF, Dynamic Modeling, Flexible Typing, Conformance

1. INTRODUCTION

Ideally, a modeling phase in a project follows a prior metamodeling phase.[1] Thus, it is usually assumed that metamodels are available before the actual modeling process starts. Unfortunately, this is not always the case, because metamodels may be incomplete even after the modeling phase has already started. It is also not the case that metamodels stay immutable after they have been defined. In the context of Enterprise Architecture, rapidly evolving metamodels, and metamodels that change after the models have been completed, are more the rule than the exception. Section 2 will discuss this point briefly in order to motivate the work presented in this paper.

It is thus necessary to have tools that are able to modify the metamodels as a result of modeling actions. This is the opposite situation of the normal *first metamodel, then model* strategy. However, none of this is easy to achieve in current tools. One reason is that they are based on a *strong conformance* relation, which usually has to be permanently guaranteed. By strong conformance, we mean that each model must conform in every moment, to the structure and restrictions imposed by the metamodel. This usually means that elements in models must be instances of types defined in correspondent metamodels; relationships between elements in models must be instances of relationships between types in correspondent metamodels, subject to the cardinality rules described there; and that element's attributes must be the all and only those defined for the corresponding types in the metamodels. Another reason is that metamodels and models are not handled in the same way, and sometimes this is not even done with the same tools. Thus the manipulation of metamodels cannot be done in a way as dynamic as the way in which models are manipulated.

All of this happens, for example, in the case of the EMF framework [1]. This case is particularly problematic because of two main reasons. On the one side, it is the most prominent framework in the modeling community and there is a large and growing number of projects and tools that depends on it. On the other side, EMF puts the strongest requirements on the metamodels: they have to be completely known before any modeling can be done; problems of non conformity are labeled as `errors`, not as `warnings`, and require immediate resolution; and changing metamodels after models have been created requires additional transformations and migrations. On top of that, after metamodeling is completed there is usually a code generation phase, which makes metamodels even more static.

[1]In this paper we will refer with *modeling* to the activities for creating models, and with *metamodeling* to the activities for creating metamodels

The aforementioned problems are summarized in figure 1. A model m_i is initially conformant to a metamodel MM_i. Later on, some changes Δ' are introduced in m_i, converting it into model m_i' and breaking the conformance to MM_i. Normally, tools require that changes are applied to m_i' in order to recover the conformance. Instead, we would like to have the possibility to easily create a metamodel MM_i'.

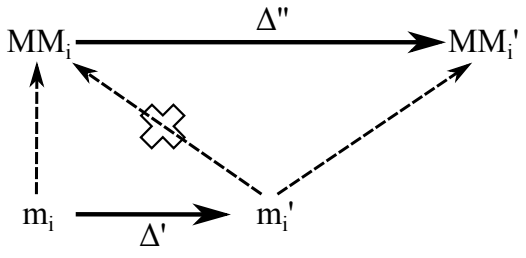

Figure 1: Problems with conformance

In this paper we present the strategy that we are applying in order to build a tool set that handles dynamic metamodels in a powerful way. This strategy is based on making a distinction between *linguistic* conformance and *ontological* conformance, following the ideas of Kühne [7]. Furthermore, the strategy makes use of a generic intermediate metamodel called GIMM (Generic Intermediate Metamodel), which is immutable and enables us to work around the limitations of EMF. The strategy has already been applied to the creation of a graphical GMF-based model editor which is independent of the domain metamodel. This editor is capable of identifying problems of conformance to the domain metamodel, and of providing solutions that work at the model or the metamodel level.

The rest of the paper is structured as follows. Section 2 presents a motivation for this work in the context of Enterprise Architecture projects. Section 3 presents in detail our solution strategy and the GIMM metamodel, and then section 4 presents the graphical editor that we created. Section 5 briefly presents some related, previous work. Finally, section 6 presents the conclusions to the work and the work that we are currently doing to improve the solution.

2. A MOTIVATING EXAMPLE

One of the goals of Enterprise Architecture (EA) projects is to create models of an enterprise that can be used to analyze its structure, and especially the relations between business, informational, and technological elements. Because of the plurality of concerns of EA models, not only the models tend to be large and complex, but also the metamodels. Unfortunately, these metamodels cannot be easily reused from one project to the next because they need to be adjusted for the enterprise under study, scope, interests, and resources available for the project. This adjust often requires a lot of time and effort, and it usually ends up in excessive costs for the projects.

The first phases of several EA frameworks and methodologies (e.g., TOGAF[8]) involve activities directed at creating the metamodel for the project. For example, some of these activities analyze the scope of the project, identify the important elements to consider, and identify the relations between those elements. This constitutes the metamodel for the project, since it will provide the underlying structure to organize all the information that will be gathered and produced during the run of the project.

On top of this, metamodels are not completely fixed from the start of the projects. They are frequently adapted as the project advances because new valuable information is found, or because some elements initially included are now considered irrelevant to the project or to the enterprise, or because the focus of the project changed. The forces that result in these changes to the metamodels can thus come from two directions. In the first one, there is a necessity to include certain aspects in a model, which the metamodel does not allow; therefore, the metamodel has to be adapted to support this new information. In the second one, the direction or scope of the project has changed, and the corresponding changes are introduced into the metamodel; as a result, the models previously created have to be made conformant to the new metamodel (this is called co-evolution as it is not something that we address directly in this paper).

These two situations define what we call *Co-Creation* of EA models and metamodels. Unfortunately, the tools available to handle modeling and metamodeling for EA projects do not properly support this co-creation, thus creating extra difficulties to the modelers that participate in the project. The solution frequently adopted is to use tools that handle non-structured information, especially MS Word and Excel, but this results in many other problems such as lack of consistency and expensive maintenance.

In the rest of this paper we present a strategy to offer proper support for model and metamodel co-creation. This strategy has been already implemented in a graphical editor, but we expect that it will be adopted by modeling tools in many different context, and especially by modeling tools in the context of EA projects.

3. DYNAMIC TYPING IN EMF MODELING

Using the Kühne's terminology, a metamodel can provide both an ontological and a linguistic framework for model creation. As an *ontological framework*, a metamodel describes what elements of the reality are represented by model elements, and what are the valid ways to relate them. As a *linguistic framework*, a metamodel defines the primitives to describe the models, their elements, and their relationships. In respect to the former perspective, model elements are *ontological instances* of the types described in the metamodels; but in respect to the latter perspective, model elements are *linguistic instances* of the types described in the metamodels. These two perspectives are complementary and they are both necessary in order to have models with semantics attached to them.

The problem with EMF is that it combines the *ontological* and the *linguistic* aspects: ontological and linguistic conformity are validated at the same time, using the same artifacts. Thus, it is impossible to create a model that conforms to a metamodel from one perspective and not from the other. While this is not necessarily a bad thing, the technological complexity associated to handling the linguistic perspective in EMF has had consequences on its ontological perspective.

In particular, EMF uses a generation-based technique to create the framework of classes to define and validate models. This has benefits for the performance of EMF-based applications, but it is very static and it is responsible for the impossibility of changing metamodels at runtime.

We now present the strategy that we have designed to overcome these limitations and allow the construction of EMF models using metamodels selected or created at runtime. This strategy separates, as much as possible, the ontological and the linguistic aspects of model construction and validation. As a result, it is possible to work around the restrictions imposed by EMF's architecture, and maintain basic compatibility with EMF-based tools.

The core of the solution presented is an intermediate metamodel called GIMM, which provides a basic linguistic framework for the definition of models. This metamodel (see figure 2) includes only those elements that are necessary to describe a basic model, and it was inspired on the subset of UML that serves to describe object diagrams; therefore, it does not include types usually encountered in metametamodels such as *classifiers*. We decided not to use the Ecore Metamodel as intermediate metamodel for simplicity.

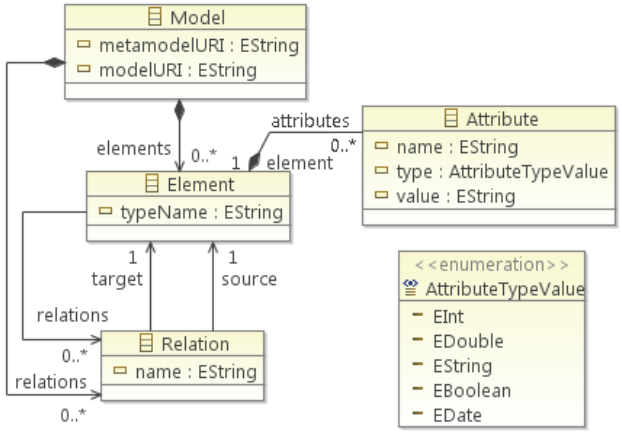

Figure 2: GIMM Metamodel

This metamodel should be straightforward to understand. The root of GIMM is the type called `Model`, which serves as the container for all the other elements. The types `Element` and `Relation` serve to represent, respectively, the element instances that appear in a model, and the relationships between them. Each element in a model has an attribute called `typeName`, that serves to relate the element to a type in the domain metamodel. Likewise, relations have names that serve for the same purpose. The type `Attribute` serves to represent the actual values of attributes of the elements contained in a model, each Attribute instance has a name, datatype and value.

Furthermore, in the current version of GIMM, attribute types may only be integers, doubles, strings, booleans, or dates, which are treated through an *enumeration*. As a result, this decision permitted us to determine a list of attribute types that can be potentially adjusted without impacting dramatically the handling of these types in the tool.

GIMM is used by means of the traditional EMF mechanisms. Thus, a framework of classes (`EClass`) based on this metamodel is generated and is used for the construction and validation of the models. This only covers linguistic conformance, since GIMM is generic and it does not have any information about the domain.

The second element of our solution strategy is a dynamic validation engine. This engine is capable of verifying the ontological conformance of any GIMM model with respect to any dynamically loaded domain metamodel. In order to do this, the engine performs several types of checks, among which compares the *typeName* of the model elements against the types defined in the domain metamodel. The engine also checks that the attributes associated to each valid element have the names and types defined in the corresponding type in the domain metamodel. The relations are also checked to see that they are valid instances of the relations defined in the domain metamodel.

By means of these two elements of the solution, we are capable of having static linguistic conformance validation, and dynamic ontological conformance validation. The only limitation is that, from a technical point of view, the model constructed is an instance of the GIMM metamodel. Nevertheless, as we show in the next section, this model contains all the information necessary to transform it into a model that conforms to the domain metamodel.

4. A DYNAMIC EMF EDITOR

The strategy described in the previous section has been implemented in a graphical editor based on EMF and GMF [3]. On the one hand, this editor serves to create models that conform to GIMM. On the other hand, this editor is also capable of validating the ontological conformity of the model with respect to a domain metamodel. On top of all this, the editor provides assistance to the user based on the domain metamodel (e.g., by indicating which are the valid types and valid attributes), and is capable of generating models that conform to that metamodel. Finally, an important characteristic of the editor is being also capable of modifying the domain metamodel, or dynamically adapting to changes introduced from outside the editor.

Figure 3 shows a screenshot of the editor. The left hand side, shows the canvas to create models conformant to GIMM. The appearance of this graphical editor was tweaked in order to make the diagram resemble an object diagram from UML [4]. For example, each element displays the class it belongs to (from the domain metamodel), and the slots with the values of attributes. Note that the attributes that appear in these elements are those specified in the domain metamodel, and not those specified in GIMM. On the right side of the image, an unmodified GMF graphical editor displays the domain metamodel that the model of the left is related to. On the bottom side of the image, the tool has the properties view that presents information related with the selected instance, and the problems view that presents details of the problems found in the model.

Another important characteristic of the editor is the way it handles problems. Normally, EMF marks any conformance problem as an error, and usually it is not possible to perform

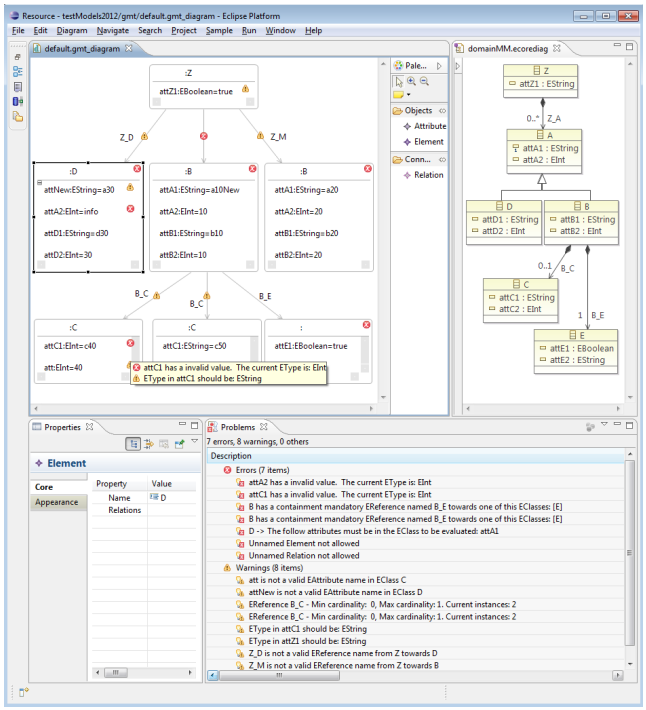

Figure 3: Screenshot of the editor

certain operations on models with errors. In our editor, we still handle linguistic conformance problems as `errors`, but ontological conformance problems are marked as `warnings`. This is possible because the former are discovered by EMF's validation elements, while the latter are discovered by our own validation engine. This subtle difference between errors and warnings facilitates the manipulation of the models and makes it possible to work with them even when they do not conform to the domain metamodel.

The canvas is the most visible component of the solution, but it is complemented by a number of other components and features. The first one is the validation engine that we already mentioned, which is responsible for verifying the conformity of a model to the domain metamodel selected. This validation is currently verified each time that a change is made to a model. When a problem is detected, it is reported as a warning and some alternative solutions are presented.

For practical reasons, the implementation of the validation engine is based on an engine for checking EVL rules [2]. The rules to check, are dynamically generated based on the structure of the domain metamodel. For example, one rule is generated to verify that the attribute `typeName` of each `element` in the model has a value that matches the name of a type in the domain metamodel. Similarly, other rules are generated to check the types and names of attributes, the cardinalities of relations, and all the other aspects that ensure conformity.

The validation engine also considers the inexistency of mandatory information in the model because sometimes it is necessary to evaluate certain information that the domain metamodel determines. For example, if the domain metamodel determines that one attribute is mandatory, then there must be a rule that validates whether the instance of that attribute exists when the associated class instance also exists in the model. Therefore, in case of inconsistency, the problem should be showed and associated with the class instance where the attribute instance should be set.

Each rule can have associated pre-conditions that determine it's evaluation; consequently, the validation engine will not evaluate certain conformity aspects until others aspects are not guaranteed. For example, the attributes of a class instance are not checked whether the class does not exist in the domain metamodel.

The EVL rules that the validation engine is based on are dynamically generated based on scripts loaded into the editor. This means that additional validation rules can be added by adding the corresponding scripts into the tool, but this is not something that is expected to be done at runtime. On the other hand, it also means that new domain metamodels can be loaded at any time and the validation engine can immediately start validating the conformance of a model against that new metamodel. This also serves to handle updates to a domain metamodel, and for this the editor has a daemon monitoring the loaded metamodel files; when one of those files is modified, the conformance rules are automatically regenerated and the validation is done using those new rules.

Another important aspect of the editor has to do with the transformations that can be applied to the various models. On the one hand, there are transformations to *import* any model and make it conform to the GIMM metamodel. In order to do this, both the model and the domain metamodel have to be loaded into the tool. On the other hand, there are the transformations to *export* a GIMM model and make it conform to the domain metamodel in the linguistic and ontological sense. This transformation is automatically applied and it is not complex. In summary, for each element in the model a corresponding element is created in the output metamodel but instead of being of the GIMM's type `Element`, it is of the type specified in the `typeName` attribute. Corresponding transformations are applied to attributes and relations. Figure 4 shows how the exported model looks like when opened with the tree model editor of EMF.

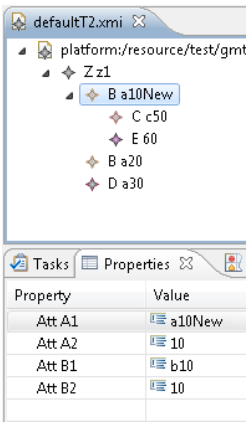

Figure 4: The ecore exported model

Finally, another feature of the editor is a set of wizards that support different modeling operations. Some of these operations are rather typical and straightforward, such as importing or exporting a model. Some others are much more complicated and are tightly related to the co-creation problem. Figure 5 shows a very representative example of this. The wizard shown in the figure is used to ask the user about the proper way to handle the addition of an attribute that is not present in the domain metamodel. Three alternatives are presented to the user: 1) select another name for the attribute from the set of attributes that are present in the metamodel; 2) add the new attribute to the corresponding type in the domain metamodel; or 3) ignore the warning and keep the model in a non conforming state. If the second alternative is selected, there is a further step on the wizard that asks for the right type in the hierarchy where the attribute should be added.

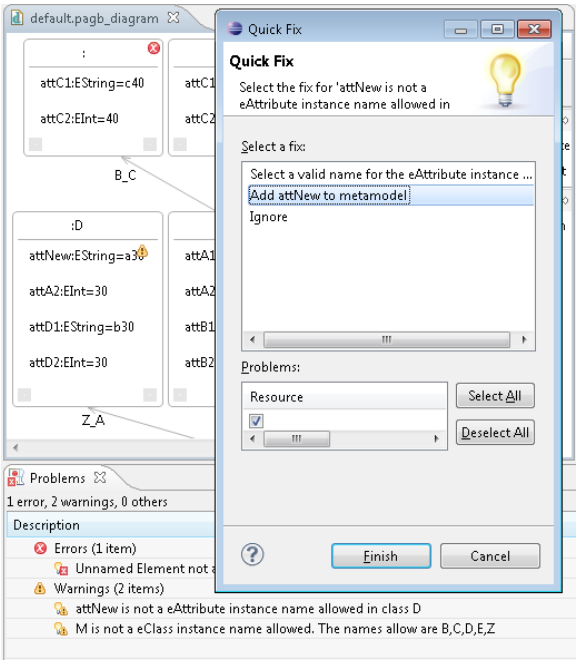

Figure 5: The wizard for an invalid attribute name

Similar to the wizards described, the editor includes other wizards which serve for two main purposes. First, they serve to steer the way in which domain metamodels are modified. Also, thanks to the monitoring services described before, changes to the metamodels are automatically discovered and rules are regenerated accordingly. The second purpose of these wizards is to ease the work of the modelers that use the tool. Instead of having to apply in a completely manual way the changes to the models and metamodels that solve the warnings, using the wizards it is possible to solve the problems in perfectly valid ways just with a few clicks. For instance, Figure 6 shows another screenshot where the model and domain metamodel shown in Figure 3 have changed after several problems and warnings have been solved.

5. RELATED WORK

In this section we briefly present some previous works that are somehow related to our own. In the first place, there is

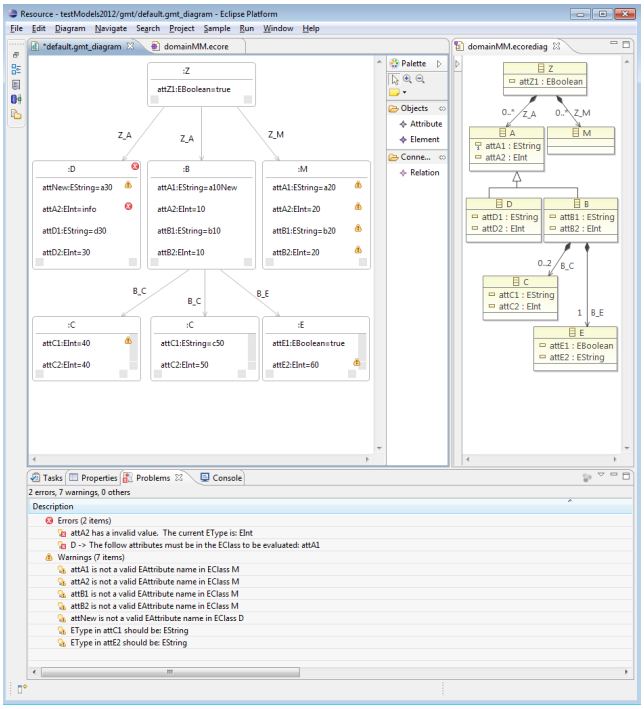

Figure 6: Screenshot of the editor with several problems and warnings in the model solved

Dynamic EMF, the part of the EMF framework that serves to create models using a programmatic interface. Although it is very powerful, it can only handle strong conformance and thus makes it complicated to change or replace metamodels at runtime. While a solution entirely based on Dynamic EMF is possible, it would require the constant application of transformations to the models under construction. Furthermore, the impact on the performance of the tool could be considerable, although we have not done the necessary experiments to compare it against our own implementation.

In [6], Gabrysiak et al. discuss how metamodels can be used in a flexible way, and they present a classification of approaches based on how dynamic are metamodels in the tools. Typically, modeling tools fall into only one of the categories they propose. However, our tool belongs simultaneously to several categories. With respect to the definition of metamodels *before modeling* (this is the first category of the classification), our tool supports *user-generated metamodels*, that is metamodels designed by the users of the tools and not by the developers of the tools. An example of this is a domain metamodel created for a particular EA project. Our tool also supports using *stencils as metamodels*, which means that some base metamodels are provided and are adapted to the particular needs of each user. An example of this is an archetypical metamodel extracted from an EA framework, which is adapted to particular projects. With respect to *modeling captured insights* (this is the second category of the classification), our tool provides support for the co-creation of models and metamodels, and also for the co-evolution of these two aspects. The third category in the classification groups those tools where the metamodel is extracted from

the model after the latter has been completed. This is not something that we are currently interested in supporting in our tool.

In [9], Ubayashi et al. present a reflective editor for the construction of models and, in particular, the construction of aspect-based models. The strategy that they present, has similar goals to the one we presented because in the end they are able to co-create models and metamodels. However, there are some fundamental differences in the approaches. Firstly, their approach is specifically targeted to aspect oriented modeling, and the only changes that can be introduced in the metamodels are extensions to model additional aspects. Secondly, their approach regenerates the editors when metamodels change. As we have seen, in our approach only the validation rules are regenerated.

The Reflective Ecore Model Diagram Editor[5] was a graphical editor based on GMF to manipulate EMF models independently of the metamodel. Therefore, the goals of this editor were very similar to those of our own one. This editor was capable of dynamically loading a metamodel, and creating models that conformed to it, but it had some restrictions related to the way it handled relations and attributes from the metamodel. On the other hand, it offered a dynamically generated tool palette with the element types obtained from the metamodel. Unfortunately the project has been abandoned since 2009 and was compatible with the Eclipse, EMF and GMF versions of the day. Because of this, and because of the difficulties to continue the work that had already been done in that editor, we developed our own solution to the problem.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed some problems related to the lack of dinamicity in model editors and the impossibility to load new metamodels at runtime. In particular, these problems occurs in EMF, which is one of the best known frameworks for the construction of model-based tools. In the paper we presented a strategy to solve this problem and we discussed how it was successfully implemented in a graphical editor based on GMF.

Although the editor is now fully functional, there are some aspects of it that are worth developing more. One aspect is to improve the appearance of the tool, and in particular of the palettes that are available to create models in the canvas. Currently, those palettes are fixed and based on the GIMM metamodel. However, we would like to be able to make those toolbars dynamic, in order to be able to configure them based on the currently loaded metamodel. Another aspect worth of being further developed is separating the two components that are currently part of the editor. The first one of those components is the graphical editor itself; the second one is the core elements that allow the dynamic manipulation and conformance validation of models. If those two components are separated, it will be a lot easier to include this tool into other tools.

Finally, there are two big ideas that we intent to pursue in order to make the editor a lot more powerful. The first one is to be able to evaluate constraints specified for the domain metamodels. Currently, this is possible if the model is exported, then the EVL validation rules are updated accordingly, then the generated model is checked, and the warning messages (if any) are mapped back into the GIMM conforming model. This strategy involves many steps frequently performed, and thus it is a candidate to be automatized. Then we state as first line of future work the creation of the mechanisms to evaluate EVL rules directly on top of the GIMM model. The second idea for future work is to evaluate the performance of the editor and optimize it.

7. REFERENCES

[1] Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/.

[2] Epsilon Validation Language (EVL). http://www.eclipse.org/epsilon/doc/evl/.

[3] Graphical Modeling Project (GMP). http://www.eclipse.org/modeling/gmp/.

[4] Unified Modeling Language (UML). http://www.uml.org/.

[5] Reflective Ecore Model Diagram Editor. http://dynamicgmf.sourceforge.net/, 2009.

[6] G. Gabrysiak, H. Giese, A. Lüders, and A. Seibel. How can metamodels be used flexibly? In *Proceedings of FlexiTools Workshop at ICSE 2011*, page 5. ACM, 2011.

[7] T. Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.

[8] The Open Group. *TOGAF Version 9*. Van Haren Pub, 2009.

[9] N. Ubayashi, S. Sano, and G. Otsubo. A reflective aspect-oriented model editor based on metamodel extension. In *Proceedings of the International Workshop on Modeling in Software Engineering*, page 12. IEEE Computer Society, 2007.