Towards a Mechanism for Controlling Meta-model Extensibility

Santiago P. Jácome-Guerrero[1] and Juan de Lara[2]
[1]*Departamento de Eléctrica y Electrónica, Universidad de las Fuerzas Armadas ESPE, Av. General Rumiñahui S/N y Paseo Escénico Santa Clara, 1715231B, Sangolquí, Ecuador*
[2]*Escuela Politécnica Superior, Universidad Autónoma de Madrid, Campus Cantoblanco, 28049, Madrid, Spain*

Keywords: Model-Driven Engineering, Meta-modelling, Meta-model Extension.

Abstract: Model-Driven Engineering (MDE) considers the systematic use of models in software development. A model must be specified through a well-defined modeling language with precise syntax and semantics. In MDE, this syntax is defined by a meta-model. There are several scenarios that require the extension or adaptation of existing meta-models. For example, OMG standards such as KDM or DD are based on the extension of base meta-models, according to certain norms. However, these norms are not *"operational",* but are described in natural language, and therefore not supported by tools. Although modeling is an activity regulated by meta-models, there are no commonly accepted mechanisms to regulate how meta-models can be extended. To solve this problem, we propose a mechanism that allows establishing norms of extensibility for meta-models, as well as a tool that makes it possible to extend the meta-models according to those norms. The tool is based on EMF, implemented as an Eclipse plugin, and has been validated to guide the extension of OMG standard meta-models such as KDM and DD.

1 INTRODUCTION

Model-Driven Engineering (MDE) is a software development paradigm that connects more closely the model to the application. In this way, the models not only encapsulate the design of the application, but are also actively used to simulate, test, verify and generate the implementation of the system to be built (García et al., 2013).

Although the modeling activity is regulated by the corresponding meta-model, there are no commonly accepted mechanisms governing how meta-models can be extended. This is because meta-models often define languages, views and services that are usually integrated in tools and are therefore less likely to need user modification (Atkinson et al., 2015). However, in some scenarios, it is common to design meta-models in order to be extended by other developersor (de Lara et al., 2014). For example, some Object Mana-gement Group (OMG) specifications are intended to be used by extending a certain part of the meta-model. This is the case of the meta-model of the Knowledge Discovery Meta-Model (KDM), and the Diagram Definition (DD). However, the ways in which these extension need to be carried out are expressed using natural language. This is error prone, more when there is no automated mechanism to check the extensions against what is specified in the standard, or guide the developer in the extension.

This situation contrasts with the well-established instantiation mechanisms of meta-models. In our view, there should be similar mechanisms to establish norms for the correct extension of a meta-model (e.g., classes to be subclassified, references to be redefined), as well as the operationalization of such norms by means of tools.

To improve this situation, the present article proposes a mechanism for the specification of rules for the extension and adaptation of meta-models, as well as a tool that allows their extension according to the defined norms. The tool has been built as an Eclipse plugin, on top of the Eclipse Modeling Framework (EMF), the de facto meta-modeling standard nowadays (Steinberg et al., 2009). The tool has been validated in different scenarios, including the definition of extensibility for the KDM (2011) and DD (2015) standards.

The rest of the article is organized as follows. Section 2 describes our mechanism for defining extensibility and adaptability in meta-models. Section 3 presents the tool support and a case study.

Jácome-Guerrero, S. and Lara, J.
Towards a Mechanism for Controlling Meta-model Extensibility.
DOI: 10.5220/0006462603820387
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 382-387
ISBN: 978-989-758-262-2

Section 4 compares with related work and section 5 ends with conclusions and lines of future work.

2 EXTENSION MECHANISM

There are situations in which meta-models are designed for the purpose of being extended. Hence, similar to object-oriented *application frameworks* (Fayad and Schmidt, 1997), these are base meta-models, from which more complex systems are derived by subclassification and redefinition. However, there is currently a lack of mechanisms to specify the way in which they can be extended.

Taking into account the need to have mechanisms that regulate how meta-models can be extended or adapted (mechanisms also referred to as "*customization*"), this section describes the approach that allows establishing norms of extensibility and adaptation of meta-models. As shown in Figure 1, this approach considers two phases.

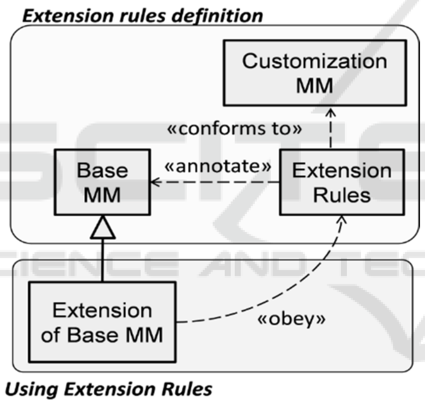


Figure 1: Definition and use of extension rules.

First, the extension rules of the base meta-model are defined. These are specified as a model, conforming to a customization meta-model (Figure 2), which *annotates* the elements of the base meta-model. In the second phase, the base meta-model can be extended according to the established rules.

We consider four types of rules (see Table 1) which permit extending, deleting, updating or creating new classes, and control reference redefinition.

It is important to note that it is common that defining the extension rules and the proper meta-model extension will be performed by different developers and will be supported by tools. This will provide guidance to the developer to extend the meta-model, as well as confidence that the extension made obeys the extension rules. Section 3 gives an overview of the developed tool.

Table 1: Supported extension rules.

Rule	Applies to	Description
Extend	Class	The tagged class is considered extensible: it is possible to add subclasses
Delete	Class	Permits tagging optional classes
Update	Class	Tags a class as "*open*": it can be added new attributes and references
New	Meta-model	Tags a meta-model as "*open*": it can be added new classes
Redefine	Reference	Governs how/if references can be redefined

In the scenario of extension of meta-models that we handle, only the extension and redefinition rules are relevant. These rules are created by instantiating the *Extend* and *Redefine* classes.

Extension rules allow extending the class (creating subclasses) of the base meta-model selected by the *custom_extend* reference. The *extensionKind* attribute declares whether the subclass must be abstract, concrete, or left to the engineer discretion. It is also possible to specify the number of subclasses allowed, using the *min..max* interval (where -1 for *max* indicates unlimited). For example, it is possible to specify whether a class must be extended exactly once (interval 1..1), optionally at most once (interval 0..1), mandatorily one or more times (interval 1..- 1), or zero or more times (interval 0..-1). It should be noted that if a class of the base meta-model does not have an associated extension rule, then it cannot be extended.

It is possible to define rules that govern the redefinition of references by instantiating the *Redefine* class. Thus, given a class C that defines a reference *ref* to a class D, we can indicate how many times *ref* can be redefined each time C is extended (through the *min..max* interval). In any case, the destination of the *ref* redefinitions must be compatible with class D. In addition, we can specify whether or not the redefinitions should be composition, or any (*compositionKind* attribute). Finally, using the *redefKind* attribute, we control the cardinality that can be assigned to each redefinition, with three possibilities:

1) *Default*. The cardinality of the redefinitions must be that of the reference *ref*.

2) *Restrictive*. The cardinality of the redefinitions must be an interval contained in *ref*. For example, if the cardinality of *ref* is 0..2, then the redefinitions can declare the intervals: 0..1, 0..2 and 1..2.

3) *Anything*. The cardinality of the redefinitions can be any.

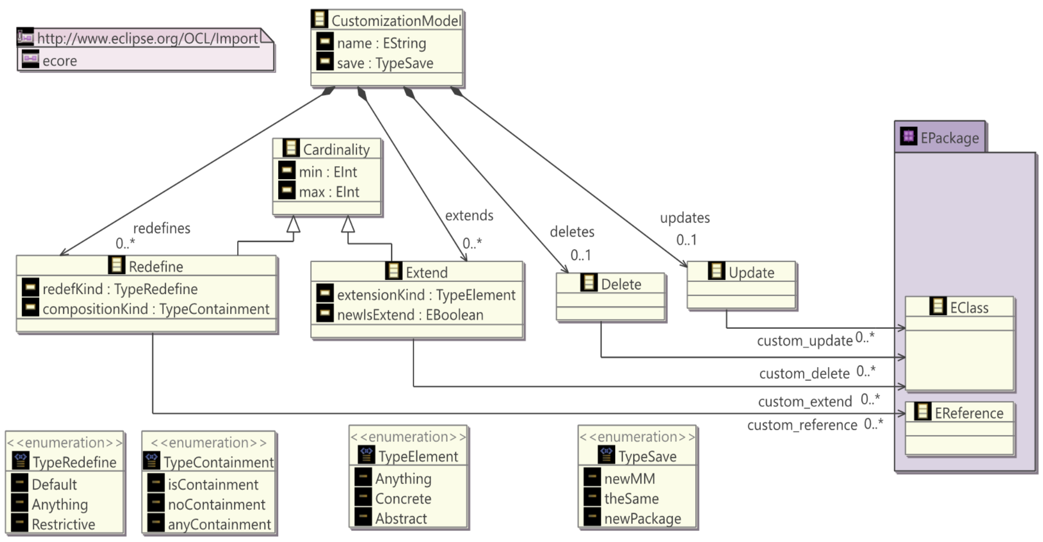


Figure 2: Customization meta-model.

When a reference *ref* is redefined by a series of references ref$_1$, ..., ref$_n$, reference *ref* is seen as a derived reference, resulting from the union of ref$_1$, ..., ref$_n$. In terms of UML, ref$_1$ ... ref$_n$ would have a subsets relation with *ref*.

We consider three options for storing the extended meta-model:
1) *newMM:* makes a copy of the original meta-model in another file for the modifications to be made on the original meta-model.
2) *theSame*: modifications are made on the original meta-model.
3) *newPackage*: the extensions are stored in a package referencing the base one.

3 TOOL SUPPORT AND CASE STUDY

This section describes tool support and we use KDM as a case study.

3.1 Tool Support

In order to make the extension approach of meta-models possible, we have designed an architecture made of a pair of complementary tools that work together. The customization architecture (Figure 3) uses the meta-model "custom.ecore" (label 1, shown in Figure 2), which specifies the operations that can be performed on the meta-model to be customized.

Then, the customization meta-model is instantiated, creating an extension and adaptation model (file with extension *.custom). This model contains the extensibility rules for the meta-model. To specify this model, the developer can use either the default EMF tree-based editor, or a textual editor that we have created with Xtext (label 2). The meta-model to be extended is loaded from a repository (label 3).

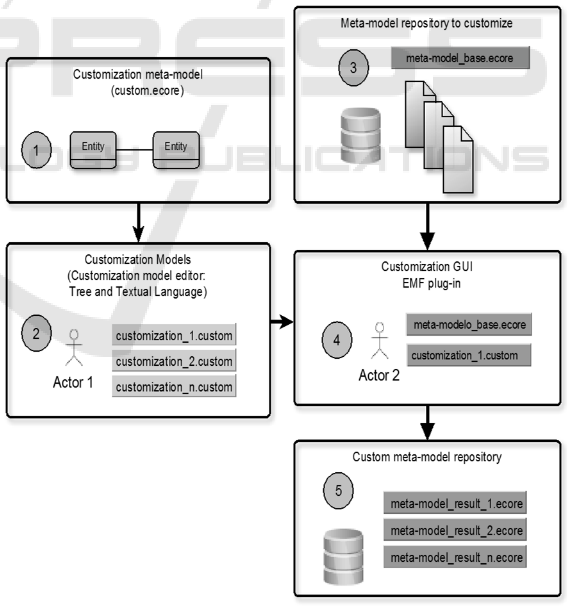


Figure 3: Customization architecture.

Once the extension rules are defined, the base meta-model can be extended according to them. For this purpose, we have created an intuitive graphical user interface (GUI), which allows executing the operations of the extension model (label 4). The custom meta-model is stored in a repository (label 5).

3.2 Case Study

In order to verify the feasibility of the proposed approach, we have defined extensibility rules on several meta-models, including standard meta-models such as DD and KDM. This section describes the extension process for KDM v1.3. The objective of this study is to define the rules of extension of the *Core* package (see Figure 4), and to use them to create one of the packages of the standard (*Code* package).

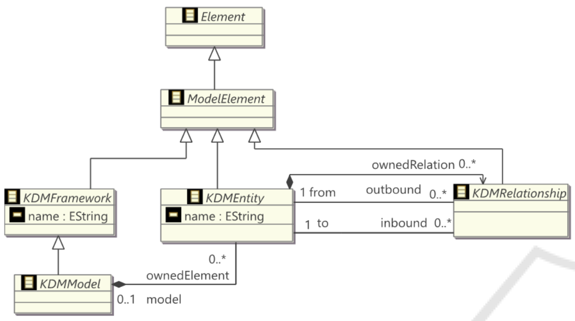
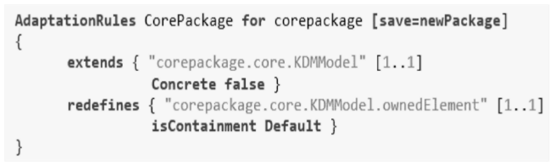


Figure 4: KDM *Core* package (excerpt).

KDM is a modeling standard in the area of software modernization. According to the standard, the KDM meta-model can be extended to represent elements and specific relations of the language, application or implementation. Its purpose is to model the various artifacts of legacy applications, such as source code or configuration files. Once the legacy system is represented in the form of models, these can be analyzed, optimized or can serve as a basis for the modernization of the application with more recent technologies.

According to the standard, extensions to this package must be made using a uniform pattern. The problem is that the KDM standard specifies these extension rules in natural language, which are error prone. That is, they are conventions that the developer can misinterpret, not understand or simply ignore. In addition, de facto modeling standards, such as EMF, do not support the redefinition of relationships (which are necessary in this case).

Figure 5 shows some of the extension rules defined with our textual language. The first line in Figure 5 specifies that the rules are for the core package, and that the extensions are to be saved in a different package.



```
AdaptationRules CorePackage for corepackage [save=newPackage]
{
      extends { "corepackage.core.KDMModel" [1..1]
                Concrete false }
      redefines { "corepackage.core.KDMModel.ownedElement" [1..1]
                isContainment Default }
}
```

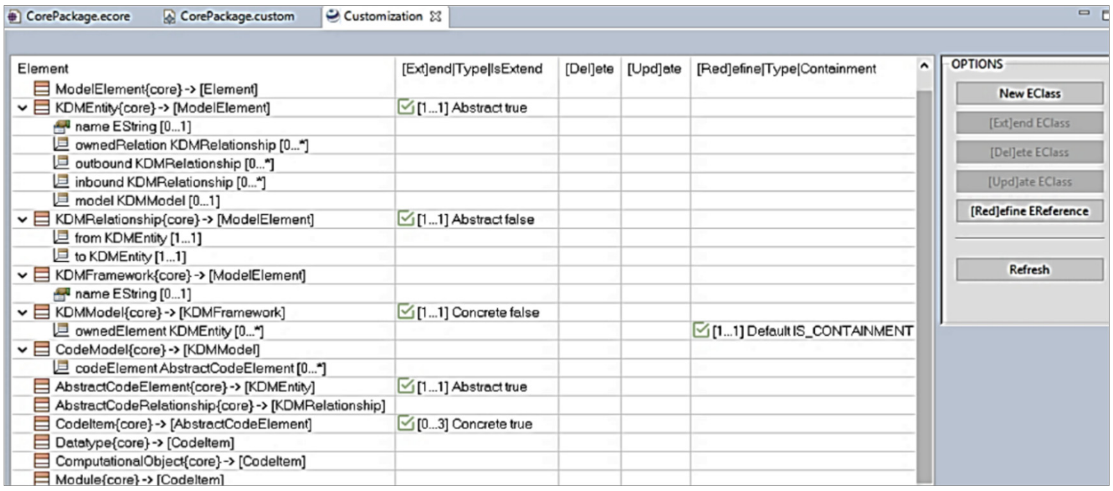
Figure 5: Some extension rules using the textual DSL.

Then the first rule (an *Extend* rule) specifies that the class *KDMModel* should be extended mandatorily (with cardinality [1..1]) by a concrete class, which in its turn cannot be further extended. The second rule (a *Redefines* rule) specifies that reference *ownedElement* should be redefined mandatorily (whenever *KDMModel* is extended) as a containment reference with same cardinality.

Once the extensibility rules are specified, they can be used through the GUI of our tool (Figure 6). This is organized into two sections, the first column of the left section shows all the elements of the base meta-model (classes, attributes and references), which can be modified according to the extension rules defined in the extension model and shown in the columns on the right, next to each element of the base meta-model. The *[Ext]end|Type|IsExtend* column specifies that the *KDMModel* class (among others) must be extended with a specific concrete Type and cannot be further extended (*IsExtend=false*).

The *[Red]efine|Type|Containment* column specifies that the reference *ownedElement* must be redefined exactly once, with default cardinality (i.e., the redefinition cardinality should be the same as the redefined cardinality). It is also specified that the reference is of the container type. It should be noted that a reference can be redefined only when the classes involved have been extended. This way, reference *ownedElement* is redefined by the codeElement reference between the *CodeModel* (which extends *KDMModel*) and *AbstractCodeElement* (which extends *KDMEntity*). The target of the reference (*AbstractCodeElement*) is compatible with the target of the reference *ownedElement* (*KDMEntity*).

The right section of the interface (panel labelled "*OPTIONS*") shows the operations in the form of buttons that are activated or deactivated depending on the extension rules specified for each element of the meta-model and which are previously defined in the extension model.



Element	[Ext]end|Type|IsExtend	[Del]ete	[Upd]ate	[Red]efine|Type|Containment
ModelElement{core} -> [Element]				
KDMEntity{core} -> [ModelElement]	✓ [1...1] Abstract true			
name EString [0...1]				
ownedRelation KDMRelationship [0...*]				
outbound KDMRelationship [0...*]				
inbound KDMRelationship [0...*]				
model KDMModel [0...1]				
KDMRelationship{core} -> [ModelElement]	✓ [1...1] Abstract false			
from KDMEntity [1...1]				
to KDMEntity [1...1]				
KDMFramework{core} -> [ModelElement]				
name EString [0...1]				
KDMModel{core} -> [KDMFramework]	✓ [1...1] Concrete false			
ownedElement KDMEntity [0...*]				✓ [1...1] Default IS_CONTAINMENT
CodeModel{core} -> [KDMModel]				
codeElement AbstractCodeElement [0...*]				
AbstractCodeElement{core} -> [KDMEntity]	✓ [1...1] Abstract true			
AbstractCodeRelationship{core} -> [KDMRelationship]				
CodeItem{core} -> [AbstractCodeElement]	✓ [0...3] Concrete true			
Datatype{core} -> [CodeItem]				
ComputationalObject{core} -> [CodeItem]				
Module{core} -> [CodeItem]				

Figure 6: Extension GUI being used to define the Code package, according to the defined extension rules.

Figure 7 shows the resulting Code package, built with the extension GUI. Altogether, the use of our approach ensures that any package extending the Core package is defined as expected by the KDM designers.

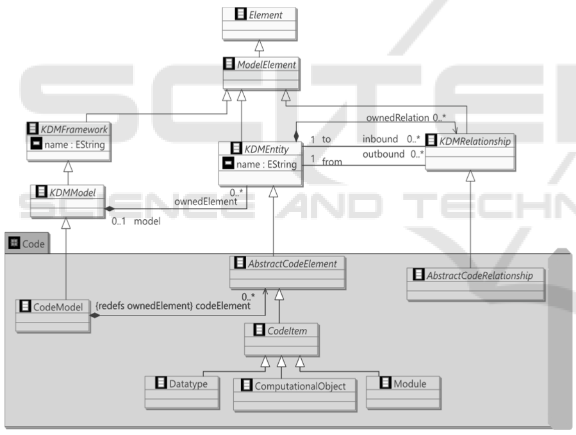


Figure 7: Resulting Code package.

4 RELATED WORK

The objective of the present work has been to propose an approach that allows controlling the extension and adaptation of meta-models. Next we compare with some related efforts in this direction.

In (Bruneliere et al., 2015) it is proposed the adaptation of meta-models through a textual DSL that allows the definition of *extensions* of the meta-model. Extensions are created as meta-model annotations by modeling experts, and can be created at the time of development of the meta-model. Unlike our approach, these mechanisms are concrete extensions to a meta-model. Our approach defines rules that must be fulfilled by any extension, and can therefore be seen as complementary to this work.

In (Braun, 2015b) it is pointed out that UML profiles provide a valuable means for adapting existing meta-models to specific platforms. The UML profiles constitute a lightweight extension mechanism where it is possible to extend the meta-model without overwriting the original elements. In (Langer et al., 2012) the philosophy of profile extension to the EMF-Ecore environment is adopted. Again, profiles can be viewed as concrete extensions to a meta-model, but not as rules that regulate their extension.

In (Braun, 2015a) a classification of extension mechanisms is provided and the concept of *hook* is considered, in order to leave open parts of a program that can be specified later in classes, interfaces or methods (Birsan, 2005), while other parts of the software remain fixed.

In (Braun and Esswein, 2015) the authors consider several MOF meta-model extension mechanisms, based on an analogy with the extension principles of the software engineering field, such as *hooks*, *aspects*, *plug-ins* and *add-ons*. An approach like ours would permit specifying the allowed extensions.

En (Atkinson et al., 2015) analyses three extension mechanisms (*built-in*, *meta-model customization* and *model annotation*), identifying strengths and weaknesses. The authors propose an alternative mechanism through multi-level modeling, which would remove the weaknesses of the previously mentioned mechanisms.

In summary, we can see a large number of works that analyze mechanisms for the extension of languages or DSLs, but there is a lack of mechanisms to define extension rules governing how meta-models can be extended, an aspect in which our work is novel and complements these existing works.

5 CONCLUSIONS

In this article we have proposed a mechanism, architecture, and a set of tools that allow to define extension rules for meta-models, as well as to make specific extensions according to the defined rules. The rules are defined by an extension model, typically constructed by the designer of the meta-model to be extended. Subsequently other engineers can use the extension rules to extend the base meta-model. Our tools guide in this extension ensuring that they obey the defined rules.

The proposed approach has the advantage that it is *non-intrusive*, and *generic*, that is, extension rules can be linked to any meta-model. On the other hand, an explicit definition of extension rules avoids the introduction of accidental errors due to the use of natural language.

We are currently improving the tool, and the expressiveness of the extension rules. Although the current rules allow expressing the extensions described in standards like KDM or DD, we will analyze other systems, to check if improvements are necessary. We will improve the tool with an assistant helping in the creation of suitable meta-model extensions. Finally, we will extend the tool to handle multi-level modeling and adaptation of DSLs.

ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish MINECO (project TIN2014-52129-R), and the Government of Madrid (project S2013/ICE-3006). As well as by Escuela Politécnica del Ejército ESPE-Ecuador.

REFERENCES

2011. *Knowledge Discovery Meta-Model™ (KDM). Version 1.3* [Online]. Available: http://www.omg.org/spec/KDM/1.3/PDF/.

2015. *Diagram Definition™ (DD™)* [Online]. Available: http://www.omg.org/spec/DD/.

Atkinson, C., Gerbig, R., Fritzsche, M., 2015. A multi-level approach to modeling language extension in the enterprise systems domain. *Information Systems,* 54**,** 289-307.

Birsan, D. 2005., On plug-ins and extensible architectures. *Queue,* 3**,** 40-46.

Braun, R., Behind the scenes of the bpmn extension mechanism principles, problems and options for improvement. Model-Driven Engineering and Software Development *(MODELSWARD), 2015 3rd International Conference on, 2015a. IEEE, 1-8.*

Braun, R., Towards the state of the art of extending enterprise modeling languages. Model-Driven Engineering and Software Development *(MODELSWARD), 2015 3rd International Conference on, 2015b. IEEE, 1-9.*

Braun, R., Esswein, W., 2015. Extending the mof for the adaptation of hooks, aspects, plug-ins and add-ons. *Model and Data Engineering.* Springer.

Bruneliere, H., Garcia, J., Desfray, P., Khelladi, D. E., Hebig, R., Bendraou, R., Cabot, J., On Lightweight Metamodel Extension to Support Modeling Tools Agility. European Conference on Modelling Foundations and Applications, 2015. Springer, 62-74.

de Lara, J., Guerra, E., Cuadrado, J. S., 2014. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM),* 24**,** 12.

Fayad, M., Schmidt, D. C., 1997. Object-oriented application frameworks. *Communications of the ACM,* 40**,** 32-38.

García, J., García, F., Pelechano, V., Vallecillo, A., Vara, J., Vicente-Chicote, C., 2013. *Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas.*

Langer, P., Wieland, K., Wimmer, M., Cabot, J., 2012. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology,* 11**,** 1-29.

Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2009. *EMF: eclipse modeling framework.*