Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

An approach to the co-creation of models and metamodels in Enterprise Architecture Projects

Paola Gómez[a] Mario Sánchez[a] Hector Florez[a]
Jorge Villalobos[a]

a. Systems and Computing Engineering Department, Universidad de los Andes, Bogotá Colombia
http://ticsw.uniandes.edu.co

Abstract The *linguistic* conformance and the *ontological* conformance between models and metamodels are two different aspects that are frequently mixed. This specifically occurs in the EMF framework resulting in problems such as the incapability to load and modify metamodels at runtime. 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.

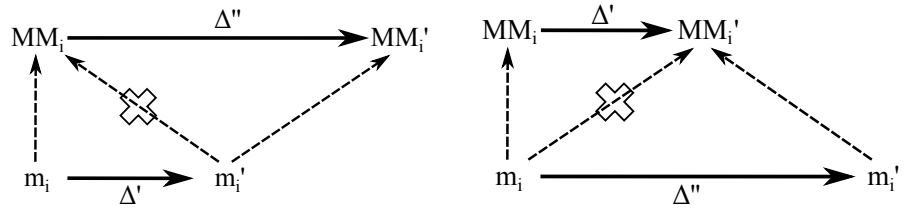
Keywords MDE, EMF, Dynamic Modeling, Flexible Typing, Conformance

1 Introduction

Ideally, a modeling phase in a project follows a prior metamodeling phase[1]; thus, making the metamodels available before the actual modeling process starts. The goal of such a metamodeling phase is to abstract the concepts and relations in the modeled domain, and define what can and cannot be expressed in the models [SCDLG12]. Unfortunately, in some cases metamodels can be incomplete or even inexistent when a modeling phase begins; they can even change afterwards. For instance, in the case of the Domain-Specific Modeling Languages (DSMLs), it is common to change the DSML as the domain is discovered [RKP12]. A similar situation exists in the context of Enterprise Architecture (EA), where metamodels evolve rapidly due to the rapid and continuous change of the current business environments [LFJU09] [Lan13]. As a result, in the EA context, metamodels that change after model creation has begun are

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

Paola Gómez, Mario Sánchez, Hector Florez, Jorge Villalobos. *An approach to the co-creation of models and metamodels in Enterprise Architecture Projects*. In Journal of Object Technology, vol. 13, no. 3, 2014, pages 2:1–29. doi:10.5381/jot.2014.13.3.a2



(a) When the model changes (b) When the metamodel changes

Figure 1 – Conformity problems

more the rule than the exception. In Section 2 we will discuss this point furthermore in order to motivate the work presented in this paper.

Given this situation, it is more desirable to have modeling strategies that are able to handle the opposite situation of the normal *first metamodel, than a model* strategy, modifying the metamodels as a result of modeling actions in an easy way. An obstacle to achieving this goal is that the current tools are based on a *strong conformance* relation, which usually has to be permanently guaranteed. By a strong conformance, we mean that each model must conform to the structure and restrictions imposed by the metamodel [RKPP10] at any given time. This usually means that a model element must be an instance of a metatype defined in the metamodels; relationships between model elements must be relationship instances between metatypes, subject to the cardinality rules as described. It also means that the element's attributes must be the only ones defined for the corresponding metatypes. Another problem is that metamodels and models are not handled symmetrically, and sometimes not even with the same tools. Thus metamodels cannot be manipulated as dynamically as the models, making the modeling process rigid [RKP12].

All of this in particular, happens in the EMF framework [EMF]. This case is very 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 very strong requirements on the metamodels: they have to be completely known before any modeling can be done [RKP12]; problems of non-conformity are labeled as `errors`, not as `warnings`, and require immediate resolution. Furthermore, changing metamodels after those models have been created, requires additional transformations and migrations. On top of that, after metamodeling is completed, there is usually a code generation phase that makes metamodels even more static.

Dynamic EMF [Bud04] which is part of the EMF framework, is very powerful and serves to create models using a programmatic interface. However, 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.

The consequences of the aforementioned problems are illustrated in Figure 1. In Figure 1a, 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. In tools based on EMF, it is necessary to fix this immediately by modifying m_i' to recover the conformance. Instead, we would like to have the possibility to introduce changes Δ'' to create a metamodel MM_i' and recover the conformance in this way.

Figure 1b shows the case when some changes Δ' are introduced in MM_i, converting it into MM_i' and breaking the conformance that m_i had with MM_i. In this case, modeling tools also require that changes are applied to m_i in order to recover the conformance. We would like to trace the effects of Δ' on the m_i, and introduce changes Δ'' to adjust m_i and converting it in MM_i' in order to recover the conformance. Automatically adjusting m_i to the changes in the metamodel, such as [HBJ09] [RKPP10], is not part of work goals presented in this paper.

The process of changing both metamodels and models incrementally has been called *co-creation of models and metamodels* [GSV13]. In this paper we present the strategy to build a toolset to support co-creation by handling metamodels in a dynamic way. This strategy is first of all based on distinguishing *linguistic* conformance from *ontological* conformance [Küh06] [DLG10]. Furthermore, the strategy identifies conformance problems to the domain metamodel and provides solutions both on model and metamodel level. In order to provide a user interface, this strategy has already been applied to the creation of a graphical GMF-based model editor called GraCoT (Graphical Co-Creation Tool), independent of the domain metamodel.

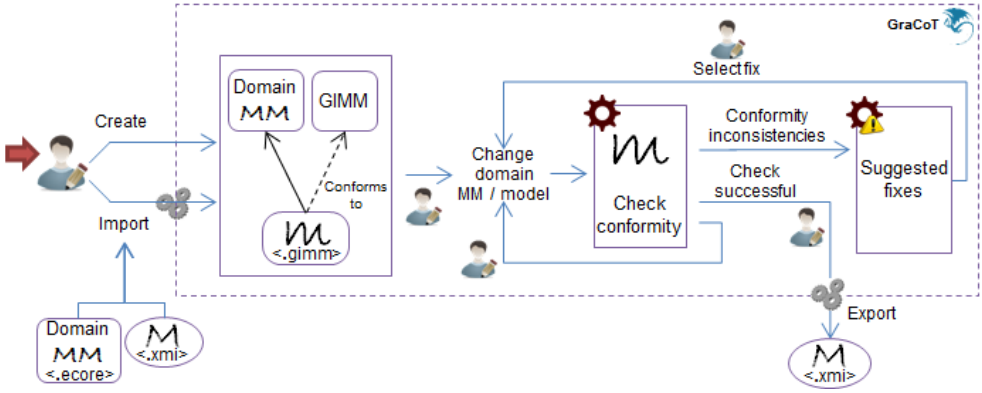


Figure 2 – Co-creation process overview

Figure 2 shows how the co-creation strategy is applied using our tool. The figure represents the general process to follow in order to create a domain metamodel and model in an incremental way. First, the user who can be the modeler or the metamodeler can create the domain metamodel and the model from scratch or import one domain metamodel and one model stored in an ecore and xmi format respectively. The domain model will be transformed in an appropriate format, which can be manipulated by GraCoT. Later, the user can apply changes to the domain metamodel or model, or check the model conformity whenever he wants. When the conformity model is evaluated, three options are available: 1) The user can change the model or metamodel again independently whether the validation was successful or not. 2) The user can select one of suggested fixes by conformity inconsistency detected in order to change the domain metamodel or model in an automatic way. In this case, the validation is also automatically applied and the validation cycle restarts. 3) The user can export the model when the validation is successful in order to obtain a standard EMF format.

The rest of the paper is structured as follows. Section 2 presents a motivation for this work in the Enterprise Architecture context. Section 3 and 4 present our solution strategy and the approach on which this strategy is based. Next, Section 5

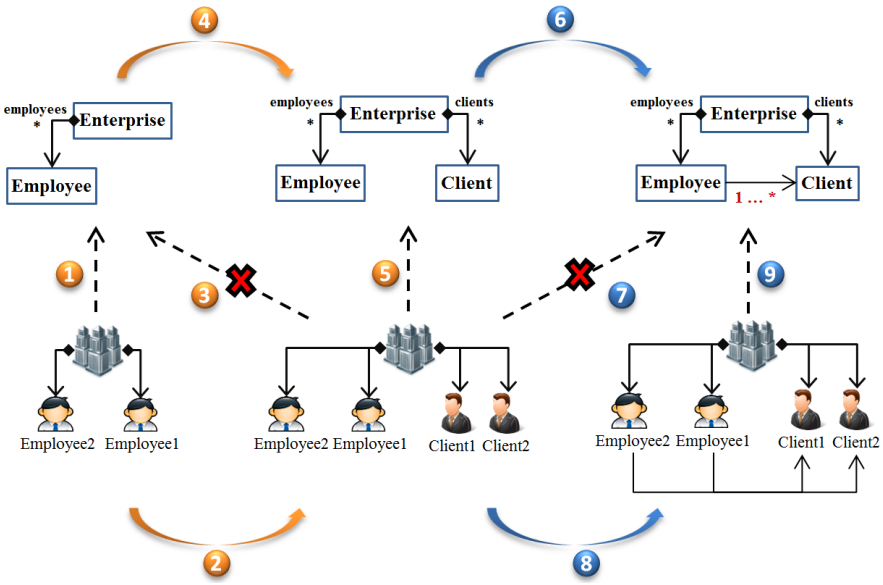


Figure 3 – Co-creation of models and metamodels example

explores the conformity validation applied to co-creation process. Section 6 explains the mechanisms to guide the user during the co-creation process. Then, Section 7 presents the graphical editor. Section 8 briefly presents some related work. Finally, Section 9 presents the conclusions.

2 Co-creation in EA

The main goal of Enterprise Architecture projects is to create a model that relates business elements to the informational and technological aspects in order to analyze the enterprise under study [CKRS12] [Ste10]. This analysis provides information that can be used to make decisions about existing problems or desired improvements of the enterprise [LFJU09] [GSV13]. The first phases of EA frameworks and methodologies (e.g., TOGAF [The09]) usually involve activities directed towards identifying elements to model their attributes, and their relations. This constitutes the metamodel for the project, and provides the underlying structure to organize all the information gathered and produced during the project. Accordingly, the metamodel can be classified as *descriptive*, because it is constructed by the enterprise observation in order to understand it [Béz05].

Because of the concern plurality and complexity of the EA models, the metamodels are typically large. 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 [LFJU09]. 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. As a result, the previously created models must be made conformant to the new metamodel.

An example of a co-creation process is illustrated in Figure 3. The model of the enterprise initially conforms to a metamodel that defines the concepts *Enterprise* and *Employee*, and establishes a relation between them; the model has initially two employees related to the enterprise. Later on, some changes are introduced in the model when two new clients are associated to the enterprise. In this moment, the conformance is broken because the metamodel did not define the concept *Client* or its relation with the enterprise. To recover the conformity, the metamodel has to be adjusted.

Afterwards, further changes are made to the metamodel: the concept *Client* is introduced, as well as one mandatory relation establishing that each employee must have at least one client. This last change breaks the conformance again, and now adjustments are required.

Unfortunately, the tools available to handle modeling and metamodeling for EA projects do not properly support the co-creation process in order to allow changes on the model or the metamodel as desired and at any time. This situation creates extra difficulties to the modelers participating in the project, because the co-creation process must be stopped to deal with technological problems in order to solve the conformity problems found.

3 An approach to support co-creation

To solve the problems previously describes, we propose a strategy based on 4 functional capabilities, which are shown in Figure 4. The main one consists of a dynamic conformity validation, whose objective is to verify the conformance of any model with respect to any dynamically loaded domain metamodel. This validation process is able to identify and classify inconsistencies between the model and the domain metamodel, and to select mechanisms helping the user to solve each problem detected.

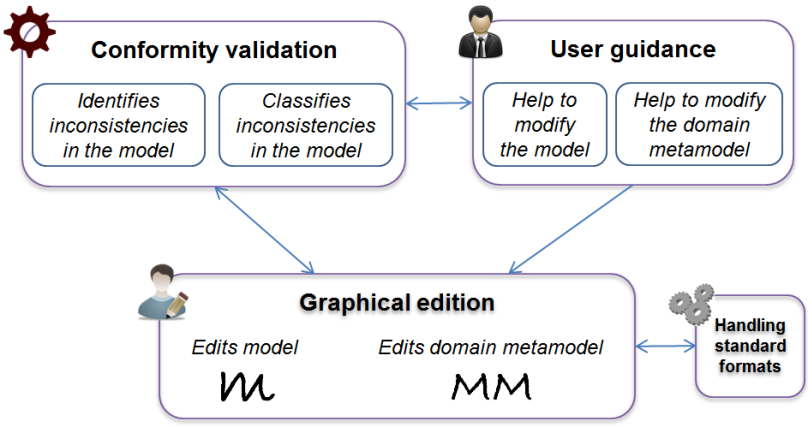


Figure 4 – Co-creation solution strategy

To solve the conformity problems identified in the model, the strategy proposes a functional capability to guide the user in this process. This functional capability helps the user to modify the model or the domain metamodel, while solving each inconsistency step by step, and applying an adequate tactic. The solution for each inconsistency (in order to recover the conformity) may involve modifications to the model and/or the domain metamodel.

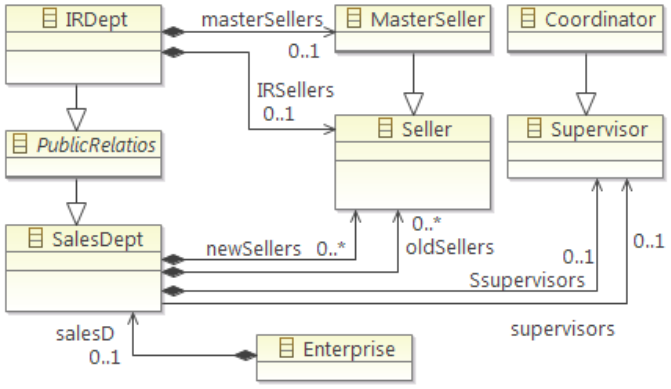


Figure 5 – Domain metamodel example

In order to provide a user interface, the strategy proposes a graphical editor to create and edit models regardless of their conformity to the domain metamodel. In addition, this editor is capable of modifying the domain metamodel, or dynamically adapting to changes introduced from outside the editor. Finally, handling of standard frameworks is guaranteed in order to manipulate different standard formats and provide compatibility.

4 Model typing

According Kühne [Küh06], a metamodel can provide both an ontological and a linguistic framework for model creation. As an *ontological framework*, a metamodel describes which information of the reality can be represented by model elements, and which are the valid ways related to them. For example, the Figure 5 depicts a metamodel that describes enterprise information related with the sales department, sellers, supervisors, and the relationships than can exist between them. As a *linguistic framework*, a metamodel defines the structural elements or primitives required to describe the models, their elements, and their relationships [DLG10]. With respect to the former perspective, model elements are *ontological instances* of the metatypes; but with respect to the latter perspective, model elements are *linguistic instances* of the metatypes. For instance, when the sellers of an enterprise are modeled, an instance of **Seller** is necessary corresponding to the domain. At the same time, an instance of some primitive is also necessary corresponding to some structural instance. These two perspectives are complementary and both necessary in order to have models with semantics.

Tools such as EMF combine the *ontological* and the *linguistic* aspects: ontological and linguistic conformity are validated simultaneously, 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. However, it is very static, and it is responsible to prevent metamodels at runtime from changing easily.

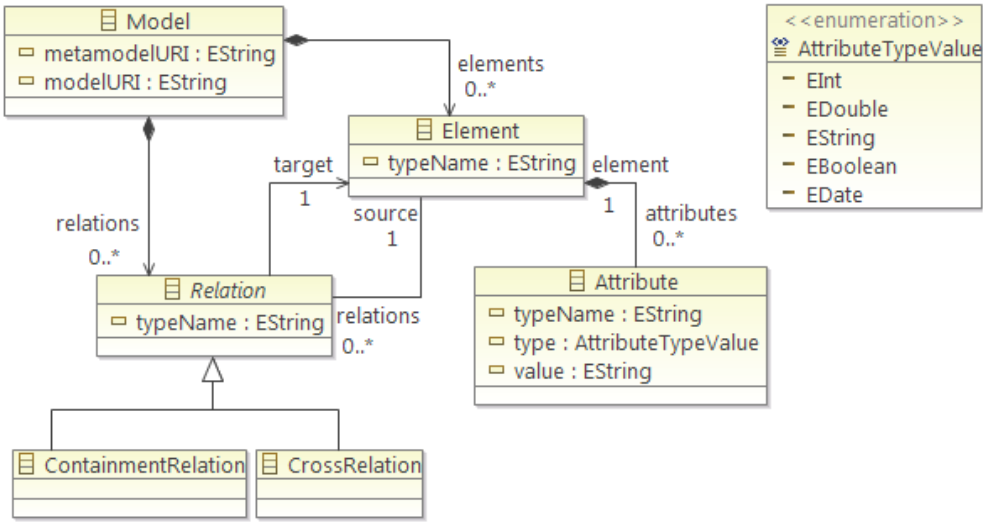


Figure 6 – GIMM Metamodel

In order to overcome these limitations and to allow the construction of EMF models using metamodels selected or created at runtime, the strategy presented in the previous section separates the ontological and the linguistic aspects of model construction and validation as much as possible. As a result, it is possible to work around the restrictions imposed by EMF's architecture, and maintain basic compatibility with EMF-based approaches.

To support this separation, an intermediate metamodel called GIMM (Generic Intermediate Metamodel) is proposed. GIMM provides a basic linguistic framework for the definition of models; this means, GIMM provides the necessary primitives to create a functional and basic model that can be handled by the different components of the strategy. This metamodel (see Figure 6) was inspired on the subset of UML that serves to describe object diagrams. Due to this, GIMM provides primitives in order to create models that follow the description of an object diagram. Therefore, we are not interested in including types usually encountered in meta-metamodels such as classifiers, as in the case of Ecore.

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 respectively represent the element instances that appear in a model, and the relationships between them. Each element in a GIMM model has an attribute called `typeName` that serves to relate the element to a metatype in the domain metamodel. Likewise, relations have names that serve for the same purpose. Indeed, the type `Relation` is classified in the types `ContainmentRelation` and `CrossRelation`, which serves to conceptually differentiate the containment property of one `EReference` in the domain metamodel.

The type `Attribute` serves to represent the actual attributes values of the elements contained in a model: each Attribute instance has a *typeName*, a *datatype*, and a *value*. In the current version of GIMM, attribute types may only be integers, doubles, strings, booleans, or dates, which are treated through an *enumeration*. The list of attribute types can be adjusted in the future without impacting dramatically the handling of datatypes.

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

In the proposed strategy, the conformity validation uses the ontological and linguistic separation in order to verify the ontological conformance of any GIMM model with respect to the domain metamodel. In addition, GIMM also provides the structure needed to guarantee compatibility with standard frameworks [EMF] through transformations to *import* any model, and make it conform to the GIMM metamodel. On the other hand, it is also possible to apply transformations in order to *export* a GIMM model and make it conform to the domain metamodel in the linguistic sense.

5 Dynamic validation of conformity

The proposal presented in this paper considers the *ontological and linguistic conformance* validation each time the model or the domain metamodel is modified. In order to support this, and thus support co-creation processes, we designed and built a validation engine that identifies conformity problems. The rest of the section focuses on presenting said engine and how EVL is used to verify both ontological and linguistic aspects.

5.1 Validating models with EVL

EVL (Epsilon Validation Language) is a language that permits to define constraints and evaluate it on models [EVL]. These constraints or invariants are grouped in contexts, which are related to specific metamodel types. Each constraint checks conditions over elements in the evaluated model, and for unsuccessful checks, the corresponding instances are marked with errors or warnings (warnings indicate that there is a possibility to fix the problem).

EVL also supports constraint dependencies, which block a constraint evaluation until all its dependencies have been validated. Because of this, a careful definition of contexts and constraints is essential for a successful execution of the validation process.

Figure 7 shows how evaluation priorities and dependencies in GIMM are related to certain elements of the metamodel that function as context for the constraints. Constraints associated to the context `Model (MC)` are the first one. They are followed by the constraints in the contexts `Element (EC)`, `Attribute (AC)`, `ContainmentRelation (CRC)`, and `CrossRelation (CrRC)`. According to these priorities, the `Attribute` instances of some `Element` instance are not checked until this `Element` instance is checked. This priority schema is the result of analyzing the dependencies between all the constraints required to adequately support a co-creation process. Henceforth, we refer to the constraints as *validation rules* because they validate conformance of the model against the domain metamodel and against GIMM.

We already said that the implementation of the validation engine is based on EVL. However, the validation rules, which are expressed as EVL constraints, are not predetermined in the engine because this would limit the capacity to use arbitrary metamodels and modify them at runtime. Instead, the engine has a number of predefined Xpand templates [XPD], which are used to dynamically generate the necessary, metamodel specific validation rules. This generation process is invoked

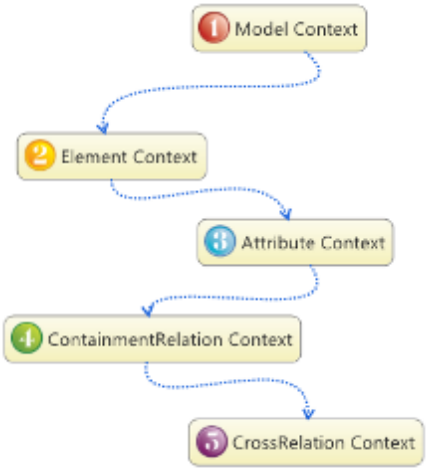


Figure 7 – Validation priorities

whenever it is necessary because the metamodel changed. There is one template for each of the validation rules that will be presented in the next section, but if necessary, these templates can be complemented with new ones. Nevertheless we are confident that the templates currently included will suffice in most cases.

Furthermore, validation rules can be classified according to the kind of context, and thus to how they are generated; they can be unique or multiple. In the first case, a single rule has to be generated and is applied in a number of places. As an example, consider the rule to verify that the attribute `typeName` of each Element matches the name of some metatype in the metamodel. Such a rule can be abstracted as the function:

$$\text{rule: } \mathit{Element} \rightarrow \{\text{true, false}\}$$
$$\text{rule(x)} \equiv \text{x}.\mathit{typeName} \in \{\ t_1, t_2, \ldots, t_k\ \}$$

Where $t_1, t_2, \ldots, t_k$ are the names of the metatypes defined in the metamodel. It can be seen that this exact rule can be applied to any element, without requiring specific generated versions.

In the case of multiple rules, it is necessary to replicate a single rule while instantiating some of its values. As an example, consider now the rule to verify that an *Attribute* a on an *Element* e is defined in the metatype t. A functional representation of such a rule could be:

$$\text{rule: } \mathit{Attribute} \rightarrow \{\text{true, false}\}$$
$$\text{rule(a)} \equiv \text{a}.\mathit{typeName} \in \{\ n_1, n_2, \ldots, n_j\ \} \wedge \text{a}.\mathit{Element}.\mathit{typeName} \in \{\ t\ \}$$

Where $n_1, n_2, \ldots, n_j$ are the names of the attributes defined in the metatype t, and this metatype corresponds with the `typeName` of e. Such a rule is only useful for the attributes of instances t, but the same pattern can be applied to any other metatype. Therefore this rule has to be replicated a number of times equals to the number of classes in the domain metamodel.

Table 1 – Problems detected by linguistic rules

RULE	Linguistic problem detected in the model	Fix
MC-LR-1	The value of the attribute *MetamodelURI* must be provided with the `URI` of the domain metamodel	✓
EC-LR-1	The `Element` instance does not have any value in the attribute *typeName*	✓
EC-LR-2	The value of the attribute *typeName* in one `Element` instance has blanks	✓
AC-LR-1	The `Attribute` instance does not have any value in the attribute *typeName*	✓
AC-LR-2	The value of the attribute *typeName* in one `Attribute` instance has blanks	✓
AC-LR-3	The value of the attribute *typeName* in one `Attribute` instance starts or ends with comma	✓
AC-LR-5	The value of the attribute *value* in one `Attribute` instance does not correspond with the type specified in the attribute *type*	✓
CRC-LR-1	The `ContaintmentRelation` instance does not have any value in the attribute *typeName*	✓
CRC-LR-2	The value of the attribute *typeName* in one `ContaintmentRelation` instance has blanks	✓
CRC-LR-3 *	Two `ContaintmentRelation` instances associate two `Element` instances with opposite direction	✗
CrRC-LR-1	The `CrossRelation` instance does not have any value in the attribute *typeName*	✓
CrRC-LR-2	The value of the attribute *typeName* in one `CrossRelation` instance has blanks	✓

5.2 Validation rules

In order to properly validate models both from the ontological and linguistical perspective, we have defined 31 validation rules and dependencies between them. These rules were inspired by the following sources: the set of validation rules that EMF applies to models [EMF]; the requirements that define the conformance of a model to the domain metamodel; and works that formalized conformance constraints in Ecore [PCP09] [PC10] or proposed strategies to re-establishing models conformance [KSP11] [SKE+14].

Out of the 31 total rules, there are 12 *Linguistic Rules*: 1) check whether the model is properly constructed with respect to the GIMM or not; 2) and satisfy some additional restrictions required to guarantee a successful ontological validation. These rules e.g check that the attribute *metamodelURI* of the model is set with a valid domain metamodel. They also check whether the values of the attributes *typeName* are properly formed (no blanks, no symbols, etc.) or not. *Linguistic Rules* are described in Table 1 where each one has an identifier divided into three parts: the first part indicates the context to which the rule belongs (e.g. `MC` for `Model Context`); the second part indicates whether the rule is linguistic (LR) or ontological (OR); and the last part is a consecutive rule. In addition, some rules analyze the absence of mandatory information in the model, and the presence of extra, non-required information. These rules, which are marked with $*$ in Tables 1 and 2, are also called *existential rules*.

On the other hand, the 19 ontological rules, which are described in Table 2, are used to evaluate the conformance with respect to the domain metamodel. A rule

Table 2 – Problems detected by ontological rules

RULE	Ontological problem detected in the model	Fix
MC-OR-1 *	The domain metamodel has several root EClasses	✗
MC-OR-2	The value of the attribute *MetamodelURI* is different from the `URI` of the current domain metamodel	✓
MC-OR-3 *	The model does not have any instance conforms to the root EClass in the domain metamodel	×
EC-OR-1	The value of the attribute *typeName* in one `Element` instance does not match with any EClass name in the domain metamodel	✓
EC-OR-2 *	The `Element` instance has several owner instance	✗
EC-OR-3 *	There are several `Element` instances that match with the root EClass in the domain metamodel	✗
EC-OR-4 *	The `Element` instance does not have associated the `Attribute` instances required	✗
EC-OR-5 *	The `Element` instance does not have associated the `ContaintmentRelation` instances required	✗
EC-OR-6 *	The `Element` instance does not have associated the `CrossRelation` instances required	✗
EC-OR-7 *	The `Element` instance does not have an owner instance	✗
AC-OR-1	The value of the attribute *typeName* in one `Attribute` instance does not match with any `EAttribute` name of the correspondent EClass in the domain metamodel	✓
AC-OR-2	The value of the attribute *type* in one `Attribute` instance does not match with the EType in the correspondent EAttribute in the domain metamodel	✓
AC-OR-3	The value of the attribute *typeName* in one `Attribute` instance is not defined, but the correspondent EAttribute in the domain metamodel has a default value	✓
AC-OR-4	The quantity of values specified in the attribute *value* for one `Attribute` instance is lower than the lower bound in the domain metamodel or greater than the upper bound in the domain metamodel	✓
AC-OR-5*	There are several `Attribute` instances with the same value in the attribute *typeName* that belongs to the same `Element` instance	✗
CRC-OR-1	The value of the attribute *typeName* in one `ContaintmentRelation` instance does not match with any EReference name of the correspondent EClass in the domain metamodel	✓
CRC-OR-2	The quantity of `ContaintmentRelation` instances belong to one `Element` instance is lower than the lower bound in the domain metamodel or greater than the upper bound in the domain metamodel	✓
CrRC-OR-1	The value of the attribute *typeName* in one `CrossRelation` instance does not match with any EReference name of the correspondent EClass in the domain metamodel	✓
CrRC-OR-2	The quantity of `CrossRelation` instances belong to one `Element` instance is lower than the lower bound in the domain metamodel or greater than the upper bound in the domain metamodel	✓

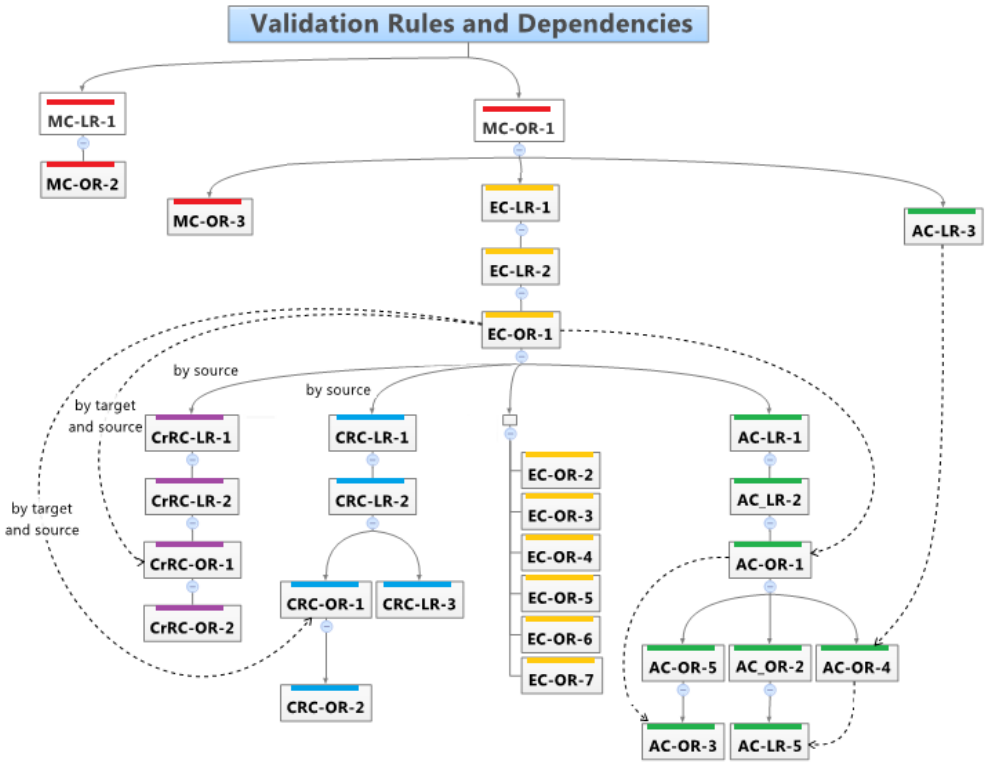


Figure 8 – Dependencies between validation rules

is classified as ontological when the verification is based on the information of the domain metamodel, independently whether this rule depends of other linguistic or ontological rules. These ontological rules use the value of the attributes in GIMM in order to match the model and its elements with the domain metamodel and its types. For example, the rule `MC-OR-2` checks that the attribute *metamodelURI* points to a valid domain metamodel. Other rules such as `EC-OR-1` and `CRC-OR-1` check whether the attribute *typeName* of each *Element* and *Relation* point to a valid metatype in the domain metamodel or not.

Validation rules may or may not offer solutions to fix the problems detected. When a solution exists, these rules mark the problematic instance with a `warning`; when there is no solution, the instance is marked with an `error`. The third column of Tables 1 and 2 uses the symbols ✓ and ✗ to indicate if the rule provides some solution to the problem detected or not.

The proposed dependencies between validation rules are presented in Figure 8. These dependencies ensure that validations can be successfully performed, even though they do not have to be defined in the same context. Figure 8 uses the same identifiers as Tables 1 and 2.

In order to illustrate these dependencies, it is possible to appreciate in the Figure 8 that the rule `AC-OR-1` requires the previous validation of rule `EC-OR-1`: `AC-OR-1` validates whether the `typeName` of one `Attribute` instance matches with some EAttribute name of the corresponding EClass in the domain metamodel; however, this can only be validated if `EC-OR-1` is successful checked.

5.3 Writing validation rules

While using concrete examples, we now will show how validation rules are defined. For this purpose we will show how a rule is describe in an Xpand template that then is used to generate one or multiple EVL scripts. We also show how some validation rules also includes fixes, which correspond to valid solutions that users may apply to fix the problems identified by each rule.

The rule `EC-OR-1`, called *hasRightElementName*, uses information of the domain metamodel to carry out the validation of the corresponding instances in the model; therefore, it is an ontological rule. This rule, checks whether the attribute `typeName` of the `Element entity` matches the name of some metatype in the domain metamodel. In this comparison, the rule ignores the abstract metatype names. In addition, this rule offers two solutions to the user through a quickfix wizard: *Select a valid name* and *Add EClass to the domain metamodel*.

The rule `EC-OR-1` is defined for the `Element context` using the EVL code presented in Listing 1. Note that this is not exactly an EVL script, but an Xpand template that when is expanded, results in a valid EVL script. In this code, the rule `EC-LR-2` is first validated in order to guarantee that the attribute `typeName` has not blanks. Next, the block `check` validates whether the `typeName` value of the current Element instance matches with any name of the EClass names collection provided by the domain metamodel. Please note, at line 4 the label `<EClass names in domain metamodel>`: this is a label that Xpand replaces with the proper values whenever the domain metamodel changes. Line 9 shows the message associated to the current instance whether the validation is not successful.

```
1  critique hasRightElementName {
2    guard : self.satisfies('EC-LR-2') //Rule name: hasBlanksInElementName
3    check {
4        var validNames = Collection { <EClass names in domain metamodel> };
5        if ( validNames.includes(self.typeName) ) { return true; }
6        else { return false; }
7    }
8
9    message: 'The instance (EClass) \'' + self.typeName + '\' has not a valid name.'
10
11   fix {
12     title: 'Select a valid name ...'
13     do {
14         var rvHelper = new Native('co.edu.uniandes.enar.gracot.rulesValidation.RVHelper');
15         var newName := rvHelper.changeInstanceEClassName( validNames , self.typeName);
16         if(newName <> null) self.typeName := newName;
17      }
18   }
19   fix {
20     title: 'Add EClass \'' + self.typeName + '\' to the domain metamodel ...'
21     do {
22         var rvHelper = new Native('co.edu.uniandes.enar.gracot.rulesValidation.RVHelper');
23         rvHelper.addEClassToMM(self.typeName, validNames , <root of the domain metamodel>);
24       }
25   }
26 }
```

Listing 1 – EVL code of the rule EC-OR-1

Two `fix` blocks indicate the possible solutions for the problems eventually detected. In the first fix block, one Java class provides a service which receives the EClass names of the domain metamodel and the current typeName. Later, according to the user actions, the new `typeName` for the Element instance is returned to modify the model directly by the constraint. In the second fix block, the Java class provides a service

which receives the EClass names of the domain metamodel, the current typeName, and the root EClass name of the domain metamodel. Later, according to the user actions, the metamodel can be modified using services from the constraint.

We now describe a more complex, multiple validation rule. The rule `CRC-OR-1`, called `hasRightContainmentRelationName SinceElement<X>`, is defined in the ContainmentRelation context, but it is replicated by each EClass that can be a source of any containment EReference in the domain metamodel. Before the evaluation, the rule depends on three rules: a) the rule `CRC-LR-2` (hasBlanksInContRelationName), which validates if the `typeName` value has blanks; b) the rule `EC-OR-1` (hasRightElementName), which validates the right `typeName` for the Element instances that are the source and relation target; c) and the `typeName` of the source Element instance corresponds with the source EClass name for which the rule has been replicated. Next, the rule checks three conditions: a) whether the `typeName` of the current ContainmentRelation instance matches with any containment EReference name of the domain metamodel where the source EClass name matches with the EClass name used to replicated the rule; b) whether the `typeName` of the target Element instance matches with any of the possible names provided by the domain metamodel in which it is considered the target EClass and its inheritance; c) Finally, the rule validates whether the ContainmentRelation instance `typeName` matches with any valid EReference name between the source and target selected. The rule offers two fixes called *Select a valid name* and *Add EReference to the domain metamodel*.

The EVL template for the rule is provided by Listing 2. Line 3 shows the described dependencies to the other rules mentioned above. In this template, at lines 5,8,28 and 43, `<X>` refers to the EClass name used to expand it whenever the domain metamodel is modified. The schemes of the lines 10 to 14 and 30 to 32 are repeated for all possible target EClass. In these schemes, `<1...n>` refers to each possible target EClass. Particularly, for each scheme, `<1...n>` is replaced for the same name where it appears. In addition, the template is responsible for providing the right EClass and EReference names considers all and only the valid names; this means that abstract EClasses are discarded, and inheritance structures are covered. Finally, two fix blocks indicate the possible solutions.

In the first `fix` block, at line 27, one Java class provides a service which receives the valid containment EReference names of the domain metamodel between the valid source and target, the valid Cross EReference names since source EClass, the `typeName` of the current ContainmentRelation instance and the `typeName` of the current source Element instance. Later, the new `typeName` for the ContainmentRelation Instance is returned to modify the model directly by the constraint.

In the second fix block, at line 47, the Java class provides another service, which receives the current ContainmentRelation typeName, the current source Element instance typeName, the current target Element instance typeName and the valid Cross EReference names since the source EClass. Later, according to the user actions, the metamodel can be modified using services out of the constraint.

Figure 9 shows an example of the rule `CRC-OR-1` applied on a concrete case. The model shown in the figure, whose conformity is evaluated over the domain metamodel shows in Figure 5, has one warning associated to the problem that this rule validates. In this model the ContainmentRelation instance called `sellers` does not have a valid typeName between the Elements instances called `IRDept` and `Seller`.

```
1  critique hasRightContainmentRelationNameSinceElement<X>
2  {
3    guard : self.satisfies(<CRC-LR-2>) and     //Rule name: hasBlanksInContRelationName
4        self.target.satisfies(<EC-OR-1>) and    //Rule name: hasRightElementName
5        self.source.satisfies(<EC-OR-1>) and (self.source.typeName == '<EClass <X> name>')
6
7    check {
8      if((Collection{ <All Containment EReference names of EClass <X>> }).includes(self.typeName)){
9
10       if( (Collection{ <Names of possible Target EClass <1...n >) and its
             inheritance>}).includes(self.target.typeName) {
11         if((Collection{ <Containment EReference names since EClass source towards Target EClass
               <1...n> > }).includes(self.typeName){
12           return true;
13         }
14       }
15       ...
16
17       return false;
18     }else{ return false; }
19   }
20
21   message: '...'
22
23   fix {
24     title: 'Select a valid name ...'
25     do {
26         var rvHelper = new Native('co.edu.uniandes.enar.gracot.rulesValidation.RVHelper');
27         var newName; var namesGroup = (Collection {});
28         var namesERefsNoCont=(Collection{ <All Cross EReference names of <X>> });
29
30       if((Collection{ <Names of possible Target EClass <1...n > and its inheritance>
             }).includes(self.target.typeName)) {
31         namesGroup.addAll((Collection{ <Containment EReference names since EClass source towards
               Target EClass <1...n > > }) );
32       }
33       ...
34
35       newName := rvHelper.changeInstanceEReferenceName(namesGroup, namesERefsNoCont, self.typeName,
             self.source.typeName);
36         if(newName <> null) self.typeName := newName;
37     }
38   }
39   fix {
40     title : 'Add Containment EReference \'' + self.name + '\' to the domain metamodel'
41     do {
42         var rvHelper = new Native('co.edu.uniandes.enar.gracot.rulesValidation.RVHelper');
43         var namesERefsNoCont = (Collection{ <All Cross EReference names of EClass <X>> });
44         rvHelper.addContainmentEReferenceToMM(self.typeName, self.source.typeName,
               self.target.typeName, namesERefsNoCont);
45     }
46   }
47 }
```

Listing 2 – EVL code of the rule CRC-OR-1

The EVL code generated in order to validate the valid ContainmentRelation instances associated to instances of the `IRDept` metatype is shown in Listing 3, which corresponds with the description shows in Listing 2. In this code, at line 9, valid ContainmentRelation names are provided based on the domain metamodel, which are selected based on the direct Containment EReferences of the EClass `IRDept` such as `IRSellers` and `masterSellers`, and the indirect Containment EReferences of the EClass `IRDept` due to its inheritance structure such as `newSellers`, `oldSellers`, `Ssupervisors`. Finally, the rule validates whether the `typeName` of the target Element matches to some direct or indirect target EClasses to the possible EReferences selected (lines 11, 15 and 19).

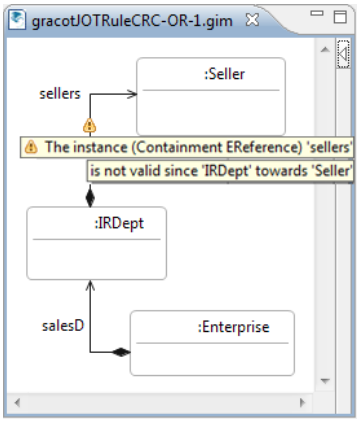


Figure 9 – Example of the validation rule `CRC-OR-1` on a GIMM model

```
1  critique hasRightContainmentRelationNameSinceElementIRDept
2  {
3    guard: self.satisfies('hasBlanksInContRelationName') and   // Rule CRC-LR-2
4           self.target.satisfies('hasRightElementName') and   // Rule EC-OR-1
5           self.source.satisfies('hasRightElementName') and   // Rule EC-OR-1
6           (self.source.typeName == 'IRDept')
7
8    check {
9      if( (Collection {'newSellers','oldSellers','Ssupervisors','IRSellers',
           'masterSellers'}).includes(self.typeName) ){
10
11       if( (Collection {'MasterSeller','Seller'}).includes(self.target.typeName) ){
12         if( (Collection {'newSellers','oldSellers','IRSellers'}).includes(self.typeName) )
13           return true;
14       }
15       if( (Collection {'Coordinator','Supervisor' }).includes(self.target.typeName) ) {
16         if( (Collection { 'Ssupervisors' }).includes(self.typeName) )
17           return true;
18       }
19       if( (Collection { 'MasterSeller' }).includes(self.target.typeName) ) {
20         if( (Collection { 'masterSellers' }).includes(self.typeName) )
21           return true;
22       }
23
24       return false;
25     }else{ return false; }
26   }
27
28   //message and fix blocks ...
29 }
```

Listing 3 – Example snippet of EVL code used by the rule CRC-OR-1 in the Figure 9

6 User guidance at co-creation

In order to guide and suggest to user different ways to solve conformity problems identified in the model, easily doing the work of modelers, the strategy proposes a functional capability focused on guiding the user through the co-creation process. Instead of having to manually apply changes to the models and domain metamodels in order to solve the problems, using automatic assistances it is possible to do so in perfectly valid ways just with a few clicks. When problems are detected, the user can use the guidance offered or not; this means that the user can choose whether he wants

to use one of the suggested assistances to recover the conformity, or he can choose to leave the model in a state of non-conformity.

A set of assistances is provided to fix the problems classified as warnings by the validation rules. These assistances are presented to the user through the quickfix options, and they are associated with each fix block of the EVL code designed for the validation rules. Therefore, these assistances are immutable due to its dependency of the validation rules, which are also immutable during the co-creation process.

When the solution is selected, the problem is solved modifying the model or the domain metamodel automatically. This solution can be applied immediately or not at all, depending on whether additional information is needed. If this occurs, this information is required through one of the 13 wizards designed for this purpose.

As in our previous work, Tables 3 and 4 show the suggested solutions provided by those linguistic and ontological validation rules that offer fix the problem detected [GSV13]. The third column of both tables indicates whether the suggested solution is interactive; this means, whether a wizard is required or not. For each suggested solution, the last column of both tables indicates whether the change is applied to the model or to the domain metamodel.

Linguistic validations focus on guaranteeing the model structure. Thus, the suggested solutions shown in Table 3 describe changes to the model that serve to recover a valid structure. If this is not achieved, ontological validation rules cannot be verified. On the other hand, these ontological validations focus on preserving the conformity between a model and the domain metamodel. As a result, the changes can be applied both to models and domain metamodels, as shown in Table 4.

Table 3 – Suggested solutions to linguistic validation rules

Rule	Suggested solution	Interactive?	Change
MC-LR-1	Set *metamodelURI* with current URI	No	Model
	Set *metamodelURI*	Yes	Model
EC-LR-1	Set an automatic Element *typeName*	No	Model
	Establish the Element *typeName*	Yes	Model
	Select a valid Element *typeName*	Yes	Model
EC-LR-2	Remove blanks	No	Model
AC-LR-1	Set an automatic Attribute *typeName*	No	Model
	Establish the Attribute *typeName*	Yes	Model
AC-LR-2	Remove blanks	No	Model
AC-LR-3	Adjust start and end commas	No	Model
AC-LR-5	Select a new valid EType	Yes	Model/ Metamodel
CRC-LR-1	Set an automatic ContainmentRelation *typeName*	No	Model
	Establish the *typeName*	Yes	Model
CRC-LR-2	Remove blanks	No	Model
CrRC-LR-1	Set an automatic CrossRelation *typeName*	No	Model
	Establish a CrossRelation *typeName*	Yes	Model
CrRC-LR-2	Remove blanks	No	Model

Note that the suggested solution to the rule `AC-LR-5` in Table 3, which checks whether the *value* of one `Attribute` instance corresponds with the *type* specified, can also change the domain metamodel. This change focuses on preserving the structure between the *type* and the valid format of the *value*, using in an `Attribute` instance. When this rule is unsuccessfully validated, the suggested solution asks the user to

Table 4 – Suggested solutions to ontological validation rules

Rule	Suggested solution	Interactive?	Change
MC-OR-2	Update *metamodelURI*	No	Model
EC-LR-1	Select a valid Element *typeName*	Yes	Model
	Add EClass	Yes	Metamodel
AC-OR-1	Select a valid Attribute *typeName*	Yes	Model
	Add EAttribute	No	Metamodel
AC-OR-2	Set the valid Attribute *type*	Yes	Model
	Set the current type as the EType	No	Metamodel
	Set a new valid EType	Yes	Model/ Metamodel
AC-OR-3	Use a default value	No	Model
	Set a default value and use it	Yes	Metamodel
AC-OR-4	Add and remove values	Yes	Model
CRC-OR-1	Select a valid ContainmentRelation *typeName*	Yes	Model
	Add Containment EReference	Yes	Metamodel
CRC-OR-2	Set current cardinality	No	Metamodel
	Modify cardinality	Yes	Metamodel
CrRC-OR-1	Select a valid CrossRelation *typeName*	Yes	Model
	Add Cross EReference	Yes	Metamodel
CrRC-OR-2	Set current cardinality	No	Metamodel
	Modify cardinality	Yes	Metamodel

select a valid *type*, which is chosen from the list of types presented in Section 4. When the selection is performed, two situations can happen at the same time: 1) if the selected *type* does not correspond with the EType provided by the domain metamodel, this is updated with the new EType; or 2) if the selected *type* does not match with the *type* used in the model, this is updated with the new *type*.

In addition, we have designed monitoring services associated to assistances that discover the changes applied to the domain metamodel automatically and update the validation rules accordingly. Subsequently, the validation is automatically applied in order to find new conformance problems. In a similar way, when the model is modified through such assistance, the model validation is also applied automatically.

Figure 10 shows an example of the quickfix used to ask the user about the proper way to handle the problem detected by rule `EC-OR-1`, which validates whether or not the attribute `typeName` of one `Element` instance match with any valid EClass name of the domain metamodel. The options provided are related with the fix blocks in the corresponding EVL template shown in the Listing 1. As a result, three alternatives are presented to the user: 1) select another name or value for the attribute `typeName` from the set of EClass names that are present in the domain metamodel; 2) add the new EClass in the domain metamodel; or 3) ignore the warning and keep the model in a non-conforming state.

In this case, the first and second alternative require an additional wizard in order to ask extra information to the user. Particularly, if the second alternative is selected, the wizard shown in Figure 11 asks the information required to add the new EClass in the domain metamodel. In this wizard, the user must provide the EReference or ESuperType information associated with the EClass, i.e. the user must choose either one ESuperType and/or one EReference. When the user selects the ESuperType option, the wizard provides one set of available names and the user must select one of them. On the other hand, if the user decides to associate the new EClass with a

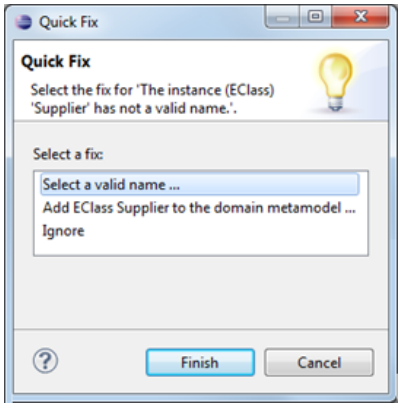


Figure 10 – Quickfix for an invalid `typeName` of an `Element` instance

new EReference, the new EClass will be the target EClass and the wizard provides one set of the available EClass names for the source EClass. Thus, the user must select one of these names and provide additional information such as name, cardinality and containment of the new EReference. In addition, the wizard validates integer formats, range of cardinality and the EReference name format. When the wizard finishes successfully, the domain metamodel is modified, the validation rules updated and the validation executed to solve the problem in the model.

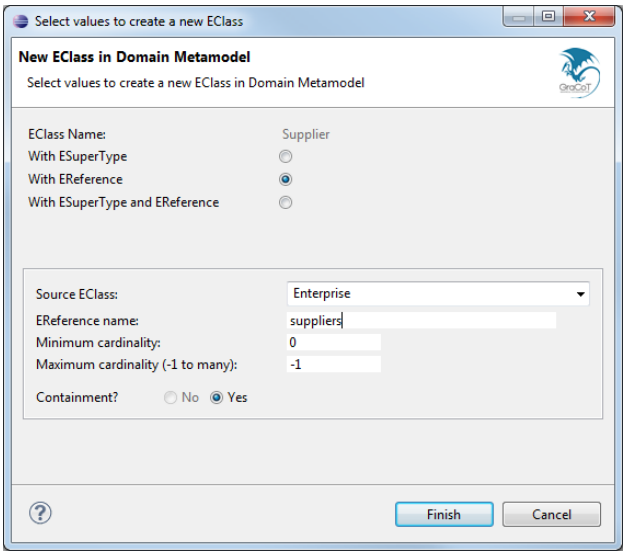


Figure 11 – Wizard that adds a new EClass 'Supplier' in the domain metamodel of the Figure 5

7 GraCoT

The strategy described in the previous sections has been implemented in a graphical editor called GraCoT (Graphical Co-creation Tool)[2], which is based on EMF and GMF technology [GMF]. GraCoT is also based on the Eclipse platform, and its plug-in architecture is composed of seven components: a) *EuGENia Editor*, which provides the graphical interface; b) *Rules Validation*, which verifies the conformance of any model respect to the domain metamodel; c) *Updates Generation*, which updates the validation rules used by the Rules Validation component; d) *Wizards*, which provides the user assistance; e) *Transformations*, which provides mechanisms to import and export to the standard xmi format; f) *Fusion Utilities*, which manipulates the domain metamodel; g) and *Utilities*, which offers generic operations. Some of these components form the core component of the tool, while others just provide extension points to allow their interaction. These technical details can be consulted in the Section 3.1 of our previous work [GSV13].

GraCoT serves to create models that conform to GIMM and it is also capable of validating the ontological conformity of the model with respect to a domain metamodel. On the other hand, GraCoT 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 handling models that conform to that metamodel. An important characteristic of GraCoT is also being capable of modifying the domain metamodel.

Figure 12 shows a screenshot of GraCoT. 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 [UML]. Each element e.g. displays the class it belongs to (from the domain metamodel), and the slots with the attribute values. Note that the attributes appearing 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 to which the left model is related. On the bottom side of the image, GraCoT has the properties view, which presents information related to the selected instance, and the problems view, which presents details of the problems found in the model.

As mentioned above, problems are marked in the canvas of the GIMM model as `errors` or `warnings` depending on whether or not they offer any solution to the user to fix the problem. This problems detection is discovered by our validation engine.

In addition to the 13 wizards related to co-creation assistances, GraCoT provides another set of 8 wizards supporting different modeling operations such as creating diagrams, selecting a new domain metamodel, exporting the model, validating the model, and establishing the GraCoT configuration.

An important aspect of GraCoT is related with the transformations that can be applied to models. On the one hand, there is a transformation to *import* any model and make it conform to the GIMM metamodel. In order to do this, both the model and the domain metamodel, which have the corresponding EMF format, are first loaded into the tool. Then, a transformation embedded in GraCoT generates the GIMM model and sets the metamodelURI to point to the right file in order to indicate the domain metamodel. On the other hand, there is the transformation to *export* a

[2]GraCoT web site: `http://gracot.virtual.uniandes.edu.co/`

Instructions for installing GraCoT are available at:
`http://gracot.virtual.uniandes.edu.co/index.php/download`

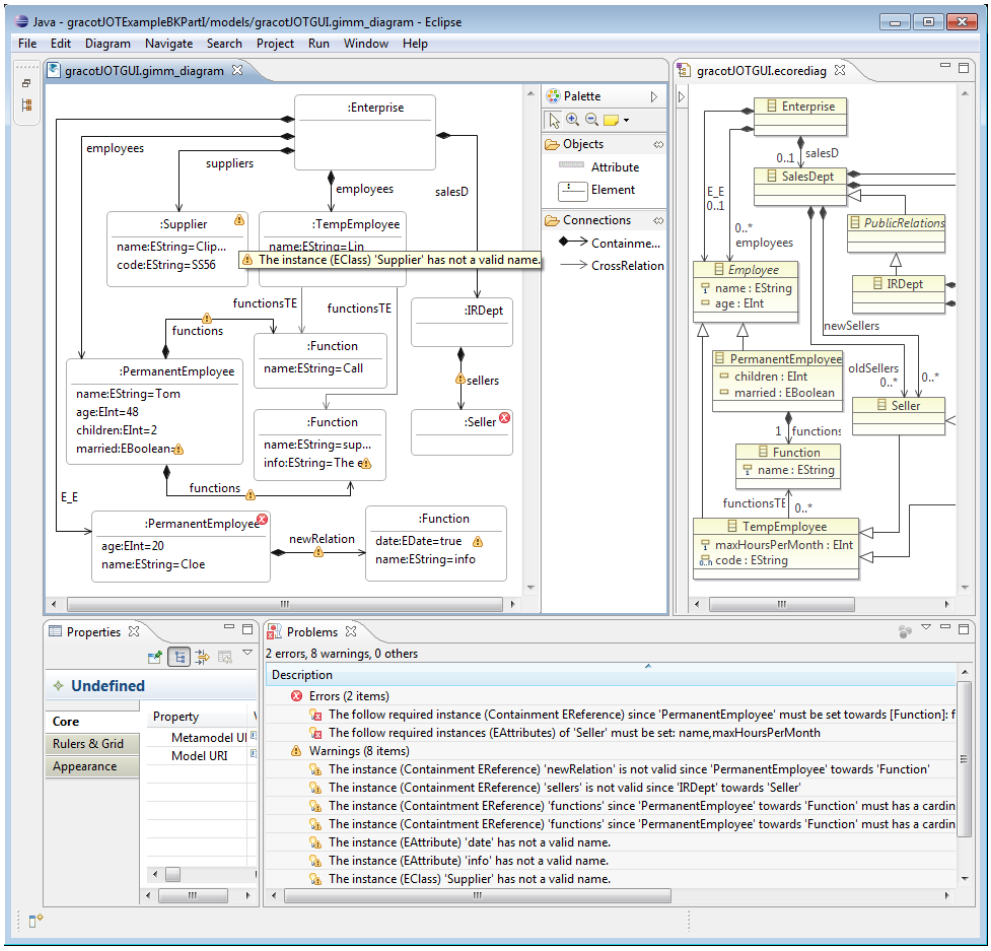


Figure 12 – Screenshot of GraCoT

GIMM model, which generates an EMF model in .xmi format and makes it conform to the domain metamodel and to the linguistic sense. This process can be done only if the GIMM model has been validated successfully.

The algorithm behind this output transformation generates an element in the output model for each GIMM model element. This is possible because of the attribute *typeName* in each `Element` instance, which is used to find the adequate metatype in the domain metamodel. The output transformation uses this mandatory attribute to create the necessary attributes and relations. Figure 13 shows a fragment of the GIMM model of Figure 12, and the corresponding exported model. In this case, they are connected by the attribute *typeName* of each GIMM model instance.

7.1 GraCoT in action

In order to evaluate our co-creation proposal and GraCoT, we have used a real scenario on a startup company called *Forever Alone*, which is a commercial scenario built for our EA Laboratory. *Forever Alone* is a social network where affiliated wholesome entertainment establishments offer exclusive products and services to single people

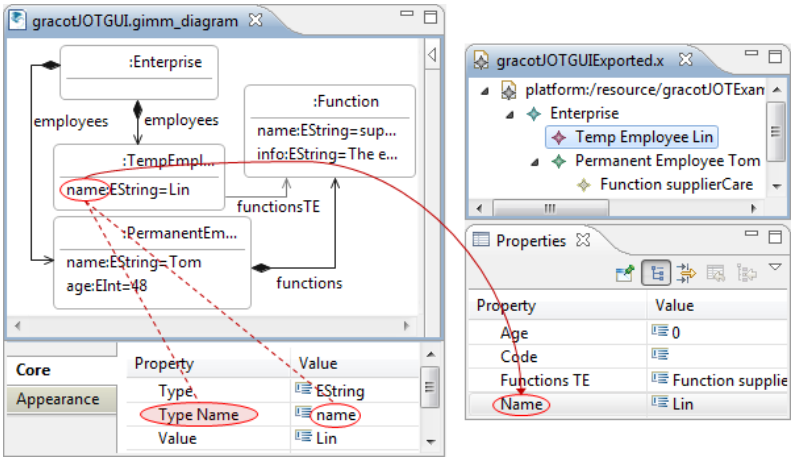


Figure 13 – A GIMM model exported

with high incomes. In addition, this single people can arrange virtual or face to face meetings through a secure technology platform that identifies and analyzes geographic location preferences and/or time available in order to ensure a pleasant encounter.

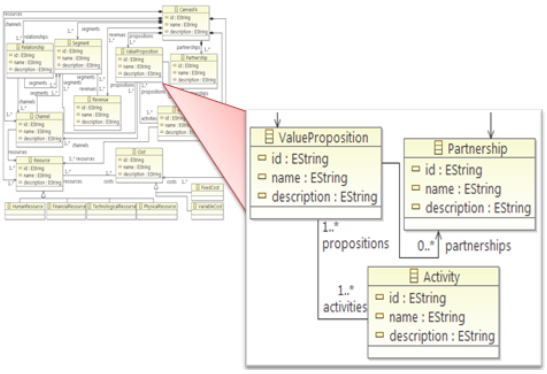
The design and the construction of this scenario are supported by a complex and large model and domain metamodel that have been built using GraCoT. Particularly, we will focus in the first design stage which consisted of modelling the *Business Canvas* in order to shows in a high level the *Enterprise Business Model* [OP10]. As a result, the metamodel of this part of the scenario incorporates the concepts and relations that the architecture group considered necessary in order to analyze the model in the future.

On the other hand, the model was built by experts and other architecture group members starting from an initial metamodel that they had to modify in order to support the information needed in the model. During this co-creation process they used the fixes that GraCot provides in order to streamline and facilitate the modeling process. Currently, this metamodel can change again in order to support new business expectations, new analysis needs, or information that was not considered previously.

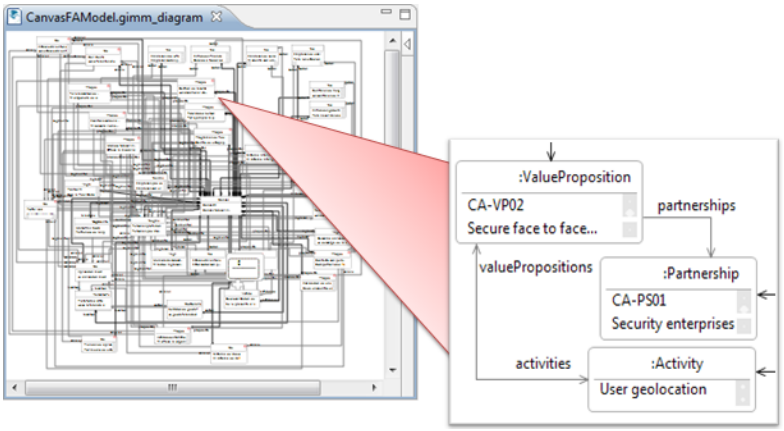
Figure 14 shows the *Business Canvas* model and metamodel built to *Forever Alone*. Figure 14a shows in background the full metamodel and the zoom, in foreground, is focused in showing just some concepts: `ValueProposition`, `Partnership` and `Activity`, and some relations: `ValueProposition` to `Partnership`, `ValueProposition` to `Activity` and vice versa. Currently, this canvas metamodel has 16 concepts and 23 relationships.

On the other hand, Figure 14b shows in background the full canvas model where it is possible to appreciate that the size and complexity is remarkable compared to the metamodel size. Currently, this *Business Canvas* model contains 69 `Element` instances, 207 `Attribute` instances and 309 `Relation` instances. In addition, the figure foreground shows a model zoom conforms to the metamodel zoom shown in Figure 14a. The zoom contains three `Element` instances whose *typeName* are ValueProposition, Partnership and Activity, and it is also possible to appreciate that `CrossRelation` instances whose `typeName` are valuePropositions and activities, are considered in order to keep the conformity with the metamodel.

Regarding the user experience, there are positive aspects to highlight and others that can be improved. As positive aspects, users expressed about the co-creation



(a) Forever Alone *Business Canvas* metamodel



(b) Forever Alone *Business Canvas* model

Figure 14 – Modeling Forever Alone

process as a practical and nice process; the possibility to modify the metamodel from the model was quite useful for them. In particular, they considered very useful to identify conformity problems and to provide fixes in order to recover the conformance. They expressed that the proposal saves modelling time, avoids manual mistakes, and gives confidence to the user when during the modelling process; he discovers that it is necessary adjusting the metamodel. With respect to GraCoT as tool, users rated it as an easy and enjoyable tool. They took just a few hours to acquire the skills to use the tool. In addition, they quickly became familiar with conformity problems identified and suggested fixes.

On the other hand, the graphical handling and the performance, in particular when the model is very large, should be improved. Users expressed a desire for a friendlier graphical handling when the model is very large because when there are a lot of instances is a bit tricky to manipulate the model. In addition, users reported an increase in the time required to validate the model as it grows.

8 Related work

In this section we briefly present some previous works that are somehow related to our own. In the first place, Gabrysiak et al. [GGLS11] discuss how metamodels can be used in a flexible way, and they present a classification of approaches based on how dynamic metamodels in the tools are. Typically, modeling approaches fall into only one of the categories they propose. However, our proposal belongs simultaneously to several categories. With respect to the definition of metamodels *before modeling* (this is the first category of the classification), our strategy supports *user-generated metamodels* that are 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 strategy 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 strategy 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 [USO07], Ubayashi et al. present a reflective editor for the construction of models and the construction of aspect-based models in particular. 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 the metamodels change. As we have seen, in our approach only the validation rules are regenerated.

The Reflective Ecore Model Diagram Editor[ECO09] 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. This editor was capable of dynamically loading a metamodel, while creating models conform to it. On the other hand, it offered a dynamically generated tool palette with the element types obtained from the metamodel. However, this editor had some restrictions related to the way it handled relations and attributes from the metamodel. In addition, the editor modified the generated code by EuGENia, and it does not propose a clear architecture, which is inconvenient for our strategy, and architecture, which we intend to decouple and enhance for scalability. 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.

In [SCDLG12], Sanchez et al. present a framework capable of creating a metamodel from model fragments that are constructed by end-users using sketching tools, such as *Visio*, *PowerPoint*, or *Dia*. This approach, called *Botton-up Meta-Modelling*, is directed to domain experts that do not know MDE but use informal drawing tools. The proposed framework transforms the models made with these informal tools into untyped model fragments which are annotated by engineers in order to indicate the actions to take when the metamodel is induced. By means of these model fragments and annotations, changes can be automatically applied on the metamodel, but only

if they do not break the conformity. Otherwise, the user is queried for additional information. The framework also provides a catalog of refactorings, and it can export the induced metamodel to EMF and MetaDepth.

The aforementioned framework differs from our proposal in several points. First of all, our proposal is targeted to modelers with sufficient MDE knowledge, or also to engineers with high capacity of abstraction. In our proposal the model does not have to be annotated in order to induce the metamodel. Also, the user can always decide how and when the model or metamodel have to be changed. Furthermore, the co-creation process does not include any automatic decisions: every change to the metamodel has to be a result of a decision made by the user, either by 1) using the assistance, or 2) by using the manual handling. A similarity between the approaches is requesting additional information from the user when the automatic change require so. With respect to the platforms, our approach is more limited, since it can only support the EMF models and metamodel.

In [DLG10], a metamodeling environment called MetaDepth is presented. This environment allows metamodeling with an arbitrary number of ontological levels and permits a dual ontological and linguistic instantiation based on a linguistic metamodel proposed by authors. Similarly to GIMM, this linguistic metamodel provides a structure for all metalevels levels. That is, models at all levels are linguistic instances of said linguistic metamodel. Additionally, each level can provide linguistic extensions, and *potency* annotations can be used to manage these multiple levels. In contrast, our proposal proposes just two levels, but the metamodel level considers two metamodels: the generic intermediate metamodel (GIMM) and the domain metamodel, which is not a linguistic instance of GIMM. On the other hand, Metadepth manages models just in the deep levels, thus suggesting a top-down strategy. Conversely, our approach supports both a top-down strategy, and a bottom-up strategy. Finally, MetaDepth is also integrated with the Epsilon language; particularly, with EVL to express constraints.

There are other approaches that allow software developers to extend existing metamodels using Profiles: In UML [UML], profiles are used to define stereotypes, that is additional metadata structures for elements in UML metamodels that define which values can be attached as tagged values in the corresponding elements in the models. For instance, a software developer can create a Persistence profile including a Table stereotype applicable to classes. This stereotype can define attributes, such as table and database names. Then, developers using the profile in an UML editor can annotate any class using that Table stereotype with concrete values for the defined attributes. The main benefits of using profiles reside in the capability of UML editors to ignore or enforce the profiles, allowing to extend metamodels without affecting existing models or tools (i.e. it is not necessary to update them). In EMF Profiles [LWW+12], the notion of profiles has been adapted to support metamodeling in EMF. This allows extending metamodels in a lightweight way, for instance by adding metamodel information without breaking the conformance of existing models. Our approach currently does not support profile extensions, stereotypes or annotations.

9 Conclusions and future work

In this paper we have discussed some problems related to the lack of dynamicity in model editors and the impossibility to load new metamodels at runtime. These problems particularly occur 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 called GraCoT, which is supported in several additional technologies to be fully functional.

To support the strategy, validation rules were developed to identify and classify the linguistic and ontological inconsistencies. In addition, assistance wizards were designed to fix these inconsistencies, and to help in adjusting the model or domain metamodel. This combination of rules and assistance wizards give GraCoT a great capability to detect and solve problems modifying GIMM models and domain metamodels through the user assistance.

Although GraCoT is now fully functional, there are some of its aspects that are worth more development. 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 allowing the dynamic manipulation and conformance validation of models. If those two components are separated, it will be a lot easier to include the co-creation features into other tools.

Finally, there are two big ideas that we intend to pursue in order to make the GraCoT a lot more powerful. The first one is to be able to evaluate constraints specified for the domain metamodels. Currently, this is only possible when the model is exported. Then, the EVL validation rules are updated accordingly. Afterwards, the generated model is checked, and the warning messages (if any) are mapped back into the GIMM conforming model. This strategy involves many frequently performed steps, and thus it is a candidate to be automatized. Finally, 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 GraCoT and to optimize it.

References

[Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005. doi:10.1007/s10270-005-0079-0.

[Bud04] Frank Budinsky. *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004.

[CKRS12] Vanea Chiprianov, Yvon Kermarrec, Siegfried Rouvrais, and Jacques Simonin. Extending enterprise architecture modeling languages for domain specificity and collaboration: application to telecommunication service design. *Software & Systems Modeling*, pages 1–12, 2012. doi:10.1007/s10270-012-0298-0.

[DLG10] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010. doi:10.1007/978-3-642-13953-6_1.

[ECO09] Reflective Ecore Model Diagram Editor, 2009. URL: http://dynamicgmf.sourceforge.net/.

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

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

[GGLS11] 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. URL: http://www.ics.uci.edu/~nlopezgi/flexitoolsICSE2011/papers/gabrysiak_flexitools_icse2011.pdf.

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

[GSV13] Paola Gomez, Mario Sanchez, and Jorge Villalobos. A tool for co-creation of models and metamodels in specific domains. In *ACadeMics Tooling with Eclipse (ACME 2013) Workshop at ECMFA, ECOOP and ECSA 2013*, 2013. doi:10.1145/2491279.2491284.

[Gua92] Nicola Guarino. Concepts, attributes and arbitrary relations: some linguistic and ontological criteria for structuring knowledge bases. *Data & Knowledge Engineering*, 8(3):249–261, 1992. doi:10.1016/0169-023X(92)90025-7.

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*, pages 52–76. Springer, 2009. doi:10.1007/978-3-642-03013-0_4.

[KSP11] Ali Hanzala Khan, Espen Suenson, and Ivan Porres. Class and object model conformance using owl2 reasoners. In Jaan Penjam, editor, *Symposium on Programming Languages and Software Tools (SPLST 11)*, volume 2011, pages 126–137. TUT Press, 2011. URL: http://tucs.fi/publications/view/?pub_id=pKhSuPo11a.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006. doi:10.1007/s10270-006-0017-9.

[Lan13] Marc Lankhorst. Introduction to enterprise architecture. In *Enterprise Architecture at Work*, pages 1–10. Springer, 2013. doi:10.1007/3-540-27505-3_1.

[LFJU09] Robert Lagerström, Ulrik Franke, Pontus Johnson, and Johan Ullberg. A method for creating enterprise architecture metamodels–applied to systems modifiability analysis. *International Journal of Computer Science and Applications*, 6(5):89–120, 2009. doi:10.1007/s11219-010-9100-0.

[LWW+12] Philip Langer, Konrad Wieland, Manuel Wimmer, Jordi Cabot, et al. Emf profiles: A lightweight extension approach for emf models. *Journal of Object Technology*, 11(1):1–29, 2012. doi:10.5381/jot.2012.11.1.a8.

[OP10] Alexander Osterwalder and Yves Pigneur. *Business model generation: A handbook for visionaires, game changers, and challengers*. NewYerk Wiley, 2010.

[PC10] Vladiela Petrascu and Dan Ioan Chiorean. Towards improving the static semantics of xcore. *Studia. Universitatis Babes-Bolyai*, LV(3):61–

70, 2010. URL: http://www.cs.ubbcluj.ro/~studia-i/2010-3/06-PetrascuChiorean.pdf.

[PCP09] Vladiela Petrascu, Dan Ioan Chiorean, and Dragos Petrascu. Proposal of a set of ocl wfrs for the ecore meta-metamodel. *Studia. Universitatis Babes-Bolyai*, LIV(2):89–108, 2009. URL: http://www.cs.ubbcluj.ro/~studia-i/2009-2/09-PetrascuChiorean.pdf.

[RKP12] Louis M Rose, Dimitrios S Kolovos, and Richard F Paige. EuGENia Live: A Flexible Graphical Modelling Tool. In *Extreme Modeling (XM 2012) Workshop at ACM/IEEE 15th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2012)*, 2012. doi:10.1145/2467307.2467311.

[RKPP10] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Model migration with Epsilon Flock. In *Theory and Practice of Model Transformations*, pages 184–198. Springer, 2010. doi:10.1007/978-3-642-13688-7_13.

[SCDLG12] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up meta-modelling: an interactive approach. In *Model Driven Engineering Languages and Systems*, pages 3–19. Springer, 2012. doi:10.1007/978-3-642-33666-9_2.

[SKE+14] Johannes Schönböck, Angelika Kusel, Jürgen Etzlstorfer, Elisabeth Kapsammer, Wieland Schwinger, Manuel Wimmer, and Martin Wischenbart. Care - a constraint-based approach for re-establishing conformance-relationships. In *Asia-Pacific Conference on Conceptual Modelling (APCCM 2014)*, CRPIT, pages 19–28, 2014. URL: http://crpit.com/confpapers/CRPITV154Schoenboeck.pdf.

[Ste10] Dirk Stelzer. Enterprise architecture principles: literature review and research directions. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 12–21. Springer, 2010. doi:10.1007/978-3-642-16132-2_2.

[The09] The Open Group. *TOGAF Version 9*. Van Haren Pub, 2009.

[UML] Unified Modeling Language (UML) Version 2.0. http://www.uml.org/. URL: http://www.omg.org/spec/UML/2.0/.

[USO07] 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. doi:10.1109/MISE.2007.3.

[XPD] Xpand. http://wiki.eclipse.org/Xpand/. URL: http://wiki.eclipse.org/Xpand.

About the authors

Paola Gómez is a PhD student in engineering at the Universidad de los Andes, Bogotá Colombia.
Contact her at pa.gomez398@uniandes.edu.co, or visit http://sistemas.uniandes.edu.co/~pa.gomez398.

Mario Sánchez is an assistant professor of the Department of Systems and Computing Engineering at the Universidad de los Andes, Bogotá Colombia.
Contact him at mar-san1@uniandes.edu.co, or visit http://sistemas.uniandes.edu.co/~mar-san1.

Hector Florez is a PhD student in engineering at the Universidad de los Andes, Bogotá Colombia.
Contact him at ha.florez39@uniandes.edu.co, or visit http://sistemas.uniandes.edu.co/~ha.florez39.

Jorge Villalobos is an associate professor and head of the Department of Systems and Computing Engineering at the Universidad de los Andes, Bogotá Colombia.
Contact him at jvillalo@uniandes.edu.co, or visit http://sistemas.uniandes.edu.co/~jvillalo.