IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 10, 2018, accepted March 21, 2018, date of publication March 29, 2018, date of current version April 25, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2821111

Controlling Meta-Model Extensibility in Model-Driven Engineering

SANTIAGO JÁCOME AND JUAN DE LARA
[1]Universidad de las Fuerzas Armadas ESPE, Sangolquí 171-5-231B, Ecuador
[2]Universidad Autónoma de Madrid, Computer Science Department, 28049 Madrid, Spain

Corresponding author: Santiago Jácome (psjacome@espe.edu.ec)

This work was supported in part by the Spanish MINECO through the project Flexor under Grant TIN2014-52129-R and in part by the R&D Program of the Madrid Region through the project SICOMORO-CM under Grant S2013/ICE-3006.

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. While meta-models tend to be fixed, there are several scenarios that require the customization of existing meta-models. For example, standards of the object management group (OMG) like the knowledge discovery meta-model (KDM) or the diagram definition (DD) are based on the extension of base meta-models according to certain rules. However, these rules are not "*operational*" but are described in natural language and therefore not supported by tools. Although modeling is an activity regulated by meta-models, currently there are no commonly accepted mechanisms to regulate how meta-models can be extended. Hence, in order to solve this problem, we propose a mechanism that allows specifying customization and extension rules for meta-models, as well as a tool that makes it possible to customize the meta-models according to such rules. The tool is based on the Eclipse modeling framework, has been implemented as an Eclipse plugin, and has been validated to guide the extension of OMG standard meta-models, such as KDM and DD.

INDEX TERMS Model-driven engineering, meta-modeling, meta-model customization, meta-model extension.

I. INTRODUCTION

Model-Driven Engineering (MDE) is a software development paradigm that connects more closely the model to the application. In this way, models not only encapsulate the design of the application, but are also actively used to specify, simulate, test, verify and generate code for the application to be built, among many other activities [1].

Models can be built using general purpose modeling languages, like the UML, but it is also common to use domain-specific languages (DSLs) addressing the needs of a particular field [2]. In MDE, modeling languages (both general purpose and domain-specific) are described through a meta-model. This is typically a class diagram containing the relevant concepts, properties, relations and constraints of a domain. A model is therefore built by instantiating the elements of the meta-model, and respecting the defined constraints. Hence a meta-model determines a (possibly infinite) set of valid models, and we say that each model in such set conforms to the meta-model.

Although the modeling activity is regulated by the corresponding meta-model, there are no commonly accepted mechanisms regulating how meta-models should be extended. This is because meta-models often define languages, views, and services that are usually integrated (and sometimes hard-coded) in tools and are therefore less likely to require user modifications [3]. However, in some scenarios, it is common to design meta-models that are meant to be extended by other developers [4]. For example, some specifications of the Object Management Group (OMG) are intended to be used by extending a certain part of the meta-model. This is for example the case of the Knowledge Discovery Meta-model (KDM) [5], and the Diagram Definition (DD) [6] standards. The ways in which these extensions need to be performed are expressed using natural language. However, this informal approach is error prone, more when there is no automated mechanism to check the extensions defined by the developer against what is specified in the standard, or means to guide the developer in the extension. This situation contrasts with the well-established instantiation mechanisms of meta-models. In our view, there should be a similar machinery to establish rules for the correct extension of a meta-model (e.g., classes to be sub- classified,

references to be redefined, enumerations that admit further literals), as well as the operationalization of such rules by means of tools.

To improve this situation, we propose a mechanism for the specification of rules for the extension of meta-models, as well as a tool that allows their extension according to the defined rules. The tool (called TACO, standing for ''a *T*ool for met*A*model *C*ust *O*mization'') has been built as an Eclipse plugin, on top of the Eclipse Modeling Framework (EMF), the de facto meta-modeling standard nowadays [7]. The tool contains an assistant that guides the developer in extending the meta-model, and has been validated in different scenarios, including the definition of extensibility for the KDM and DD standards, and other meta-models built by third parties.

This paper is an extended version of our preliminary work [8], [9], where we have improved the expressivity of our approach (e.g., to support extensibility of enumerations, and for a more fine-grained control of allowed updates), and improved tool support with an assistant to guide the developer in the meta-model extension. We present a classification of meta-model customization types and their effects, a case study using the DD standard, and an applicability study of our technique, which analyses the extension needs of existing meta-models built by third parties, identifying recurring extensibility needs.

The rest of this article is organized as follows. Section II describes scenarios where meta-model customization control mechanisms are useful. Section III presents a classification of customization types. Section IV shows our mechanism for defining allowed customizations of meta-models. Section V describes tool support. Section VI presents an evaluation of our approach. This evaluation has two parts. First, we report on a field study showing the need for extensibility in meta-models built by third parties. Second, we show a case study using the DD standard. Section VII compares our approach with related work, and Section VIII finishes the paper with conclusions and open lines for future research.

II. MOTIVATION AND USAGE SCENARIOS

Taking into consideration that meta-models are the cornerstone in the development of software with MDE [10] and that software has to evolve over time [11], it is necessary to have frameworks that provide methodological support in the evolution of meta-models. This evolution permits developing software that can adapt to new application domains or changing requirements within the domain [10].

There are several independent efforts that address the issue of meta-model extension, but standards have not yet been developed for this task [12]. Braun and collaborators [10] point out that an extension improves the expressiveness of a conceptual modeling language by introducing new constructs and properties or by refining the existing elements in order to represent concepts of specific purpose.

Next, we review three scenarios motivating the need for specifying allowed meta-model customizations, as well as mechanisms to guide the developer to perform the customization according to those rules.

A. EXTENSION OF META-MODELS

There are situations in which meta-models are designed for being extended. Hence, similar to object-oriented *application frameworks* [13], these are base meta-models, from which more complex systems are derived by sub-classification and redefinition.

For example, the DD standard of the OMG allows the formal specification of the concrete syntax of a modeling language making it possible for the tools to exchange diagrams [6]. The idea of the standard is to obtain a systematic way to exchange concrete syntax information between tools, similar to the way the models are interchanged between MOF-based tools in XMI, a specification that maps MOF to XML [14]. The abstract syntax of a graphical modeling language is typically defined with a meta-model while its concrete syntax (a diagram) is informally defined with text and figures [15], [16].

The DD standard considers an architecture that allows the specification of a *Diagram Interchange* (DI) and a *Diagram Graphics*(DG) model. DI is a framework intended for extension rather than a ready-to-use component. Extensibility allows defining graphical aspects that the user controls, such as the position of the nodes and the routing points of the lines. Hence, we can see that DI needs to be *refined* for its use with a particular modeling language (e.g., the UML). DG is used to define the graphical aspects that are specified by the modeling language (uncontrollable by the user) [15]. Both models share common elements from a *Diagram Common* (DC) model. The DD architecture expects language specifications to define mappings between interchanged and non-interchanged graphical information, but does not restrict how it is done (Fig. 1).

As noted, using the DD standard for a specific modeling language requires the extension of certain classes of the DI meta-model and the way of extending such meta-model is described in natural language in the standard. However, this can lead to errors, more when there is no automated mechanism to check the correctness extensions, or guide the developer in their construction.

B. MULTI-LEVEL MODELING AND THE TYPE-OBJECT PATTERN

In traditional meta-modeling the engineer works with two meta-levels: meta-models and models. This is the approach followed by current meta-modeling standards such as the EMF. Multi-level modeling [4] is a conservative extension of the two-level approach, supporting the use of an arbitrary number of meta-levels to describe the systems. Multi-level modeling makes it possible to obtain simpler models (with fewer elements) in some scenarios, especially when the type-object pattern arises [4].

As an example, assume we want to describe an e-commerce system, supporting the *dynamic* definition of

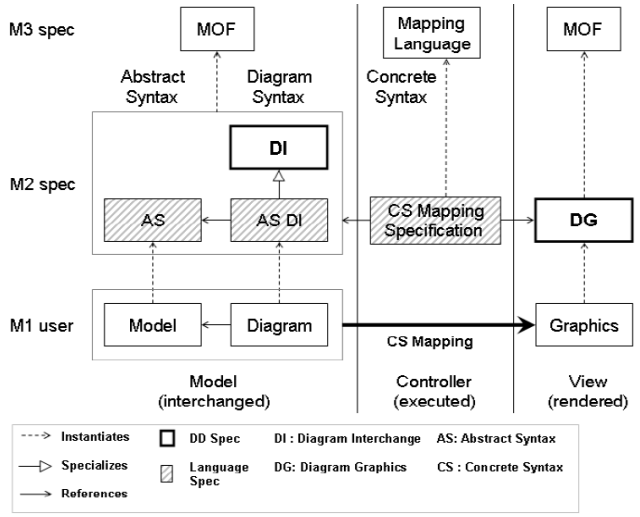

FIGURE 1. Diagram Definition Architecture, taken from [6]. For its use, the DI meta-model needs to be extended (see AS DI meta-model).

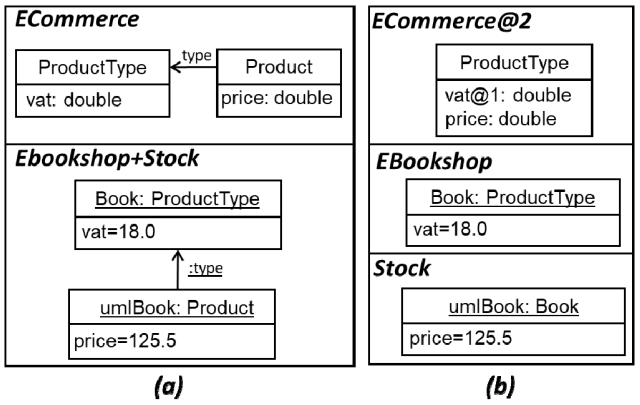

FIGURE 2. An E-commerce system modelled using the type-object pattern (a), and multi-level modeling (b).

both product types (like *Books*) and instances of these. In this domain, product types may have different value added tax (*VAT*), while instances have a price. Such system is represented in Fig. 2 (a) using the type-object pattern. The meta-model contains classes modeling product types, instances and the typing relationship between both. Alternatively, we can use multi-level modeling to represent the same system, as Fig. 2(b) shows. This solution uses three meta-levels. The top-most model does not require explicitly modeling *Products*, as these will be instances of the instances of *ProductType*. Mechanisms for deep characterization, like potency [4], are used in multi-level modeling to describe instances beyond the immediate meta-level below. The potency is a natural number or zero, which states the number of meta-levels below at which the attribute (or class) can be instantiated. In the figure, it is shown after the "@" symbol. When instantiating an element, the instance receives a decreased potency, and an element with potency 0 cannot be further instantiated. In the Figure, *ProductType* has potency 2, so it can be instantiated in the following two meta-levels. Attribute *price* has potency 2

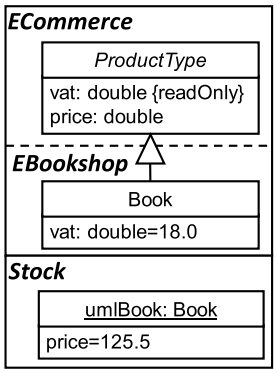

FIGURE 3. The E-commerce system modelled using subclassing.

as well – so it applies to instances two levels below – while *vat* has potency 1 and applies to instances at the next level only. As it can be noted, the resulting multi-level model has less elements than the two-level solution.

Multi-level modeling has advantages in the design of the extensibility of meta-models, since the needs for extensibility can be expressed as instantiation, for which there are standard mechanisms that describe its operation. There are several tools that support multi-level modeling, such as metaDepth [17] or Melanee [18]. However, they are not directly interoperable with plain EMF and its rich modeling ecosystem.

Multi-level modeling could be emulated within standard two-level modeling by replacing instantiation with subclassing. This would have the advantage of maintaining the compatibility with standards, like MOF, but would require enhancing the control of how extensions of a meta-model can be performed. By adding such control rules, we would like to avoid the creation of incorrect extensions, or forbidden modifications of the base meta-model.

Fig. 3 shows the E-commerce system using an extension-based mechanism. This way, *Book* is defined as a subclass of *ProductType*, instead of as an instance of it. A realization of this approach would require means to emulate the "*instantiation*" of features (like *vat*) at the meta-model level (like in *Book.vat*). Please note that, by supporting this scenario, it would be possible to migrate from the type-object pattern (Fig. 2(a)) to a solution based on subclassing (Fig. 3). One of the main rationales for using multi-level modeling or the type-object pattern (Fig. 2) is to obtain a modularity mechanism, by which users, different from the meta-model designer, can dynamically create new types (*Book* in Fig. 2). This would be achieved in the solution of Fig. 3 if the base meta-model designer would be able to define extensibility rules, which then could be used by others to define extensions like *EBookshop* in Fig. 3.

C. ADAPTATION OF DSLs

Many DSLs are not completely designed from scratch, but are created by extending or adapting a base language to a certain domain [19]. For example, designing DSLs to describe behaviour is often based on well-known languages, such as

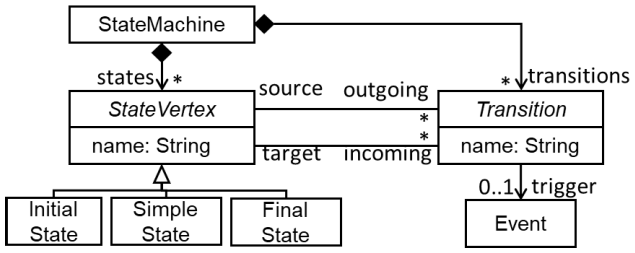

FIGURE 4. A meta-model for State machines.

state machines (see a simplified meta-model in Fig. 4), and adding domain-specific extensions.

In this scenario, in addition to extending existing elements, adaptations and simplifications (deletions) may also occur. In case of Fig. 4, for a specific domain, a designer may not need state machines with final or initial states. In addition, to adapt the State machine to a specific domain, s/he may need to update class *SimpleState* with new features, and define several subclasses of *Event*. If we need to ensure a correct reuse of associated artefacts like model transformations, a specification of elements that cannot be changed (e.g., *StateMachine*, *StateVertex*, *Transition*, *SimpleState*), optional elements that can be removed (*InitialState*, *FinalState*, *Event*) and extensible elements that can be subclassed or updated (*Event, SimpleState*) is required. While existing techniques e.g., based on product lines [20] can be used to annotate meta-model elements with presence conditions, those approaches do not support a specification of how and if certain elements can be extended (subclassed) or refined (in case of references). Hence, they are a *closed* approach to reuse (where a certain product of the product line is chose), while our proposal would lead to an *open* approach to reuse, where a base meta-model is adapted and then extended in a controlled way.

In our view, a repository of language components, with indications of how these can be reused (extended, restricted, adapted) would facilitate the task of constructing new DSLs, or, more generally, flexible reuse of artefacts associated to meta-models. In this scenario, a co-adaptation of the associated artefact is required.

Altogether, in these three scenarios, we see a need to express how certain elements in a base meta-model can be refined (for the scenarios in Sections A and B), expanded with new features or simplified (for the scenario in Section C).

In the rest of this paper, we present an approach to the description of extensibility mechanisms, focusing on the meta-model extension scenario. However, first we analyse the types of meta-model customizations and their consequences.

III. CLASSIFYING META-MODEL CUSTOMIZATION TYPES

In this section we classify the kinds of customizations that can be performed on a meta-model, and discuss the compatibility of possible existing source models with respect to the customized meta-model, and vice-versa. The types of customization are summarized schematically in Fig. 5.

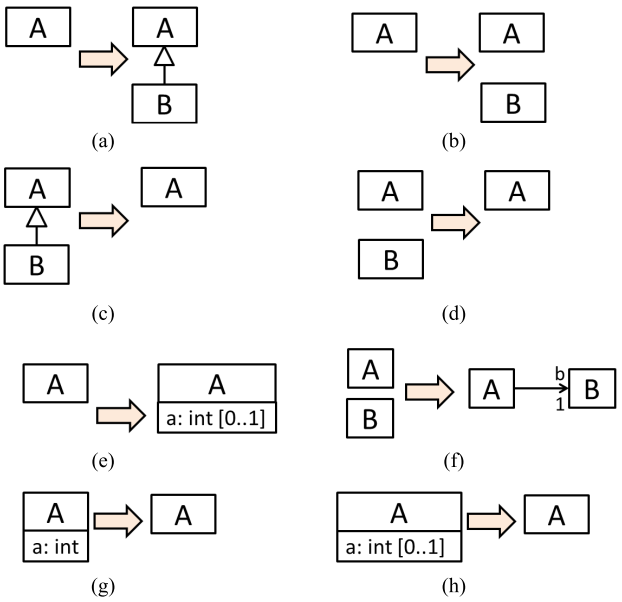

FIGURE 5. Meta-model customization types. (a) Meta-model refinement. (b) Meta-model expansion. (c) Meta-model restriction. (d) Meta-model contraction. (e) Conservative additive customization. (f) Breaking additive customization. (g) Restrictive customization. (h) Restrictive safe customization.

The first four cases refer to classes, while the last four cases involve features.

When talking about modifying the meta-model, several terms are used in the literature, including extension, customization, adaptation, variation or mutation. In the present work we use the term "*customization*" to refer to any kind of modification, while "*extension*" is a specific type of customization that adds new subclasses, or new classes to the meta-model, while leaving the existing elements intact. This distinction is useful, as extensions preserve compatibility of existing models with respect to the resulting extended meta-model, while arbitrary customizations may not preserve such conformance.

We distinguish two kinds of extensions: meta-model refinement and meta-model expansion, shown in Figures 5(a) and 5(b) respectively. A refinement adds new subclasses to existing classes[1] in the original meta-model. The new subclasses can add new features and redefine references of the original class. An expansion adds new classes that are not related by inheritance to the existing classes of the original meta-model.

As Table 1 shows, a model conforming to a meta-model, also conforms to a refined meta-model and to an expanded one (column forward model compatibility). However, a model conforming to a refined or expanded meta-model does not necessarily conform to the original meta-model (column backward model compatibility).

Following [21] we distinguish incompatibilities which can be automatically resolved (*R* in the table) and that are unresolvable automatically and require human intervention (*U* in

[1] We consider single inheritance only.

TABLE 1. Forward and backward compatibility of meta-model modification types. (Types: {E=Extension, C=Customization, PC=Property Customization}, Compatibility: {=Compatible, R=Resolvable, U=Unresolvable}).

Type	Modification	Forward model compatibi.	Backward model compat.
E	Refinement	✔	R
E	Expansion	✔	R
C	Restriction	R	✔
C	Contraction	R	✔
C	Conservative additive customization	✔	R
C	Breaking additive customization	U	R
C	Restrictive customization	R	U
C	Restrictive safe customization	R	✔
PC	Abstractize class	U	✔
PC	Concretize class	✔	U
PC	Inheritance introduction	U	U
PC	Inheritance deletion	U	U
PC	Cardinality relaxation	✔	U
PC	Cardinality restriction	U	✔
PC	Cardinality change	U	U
PC	Feature type relaxation	✔	U
PC	Feature type restriction	U	✔
PC	Feature type change	U	U
PC	Set composition	U	✔
PC	Unset composition	✔	U

the table). Both refined and expanded models can be made conformant to the original meta-model automatically. In the first case, we retype objects of the subclass to the super class (objects of type B to objects of type A in Fig. 5 (a)), and remove the additional slots and links introduced by the subclass. Similarly, for an expanded model we simply remove the objects of the added classes.

Conversely to refinement and expansion, we may have restrictions (i.e., deleting a subclass) and contractions (deleting a class with no superclass). These customizations are inverses of refinement and expansion respectively, and hence the forward and backward compatibility are reversed.

Next, we focus on class features, where we distinguish two other types of customizations: additive and restrictive. The former adds new features to existing classes in the original meta-model, while the latter deletes features of existing classes.

Figures 5(e) and (f) show two subtypes of additive changes. The first are called conservative, since they add optional features (attributes or references) to existing classes. Conservative additive customizations preserve conformity of models with respect to the updated meta-model. Models of the updated meta-model can be made conformant to the original one by removing the slots corresponding to the newly added features.

Breaking additive customizations add mandatory features to existing classes (see Fig. 5(f)). They break conformance of existing models, since these lack such mandatory features and hence cannot be correctly typed with respect to the updated meta-model. Such non-conformity is unresolvable automatically, and would require user intervention (as there may be several ways to fix an existing model). Conversely, models of the updated meta-model are not conformant to the original meta-model, but can be fixed automatically by removing the slots corresponding to the newly added features.

Restrictive customizations delete features of the original meta-model. We distinguish between removing mandatory (Fig. 5(g)) and optional features (Fig. 5(h)). Removing mandatory features makes existing models incompatible with the updated meta-model, but incompatibilities can be automatically fixed by removing the slots of the removed mandatory features. Conversely, models of the updated meta-model cannot be made conformant to the original meta-model in an automatic way, as there would be many different values that can be chosen for the slots that need to be added.

Restrictive safe customizations are those that delete optional features of existing classes (see Fig. 5(h)). These updates may cause incompatibility of existing models with respect to the updated meta-model, which can be automatically resolved, by removing the slots and links corresponding to the deleted features. Conversely, models of the updated meta-model remain conformant to the original meta-model, because only optional features are removed by this customization type.

Altogether, it can be noted that conservative additive customizations and restrictive safe customizations are inverses of each other; while breaking additive customizations and restrictive customizations are also inverses.

The presented changes are enough for our envisioned scenarios of meta-model extension and customization. However, for completeness, next we discuss additional possible meta-model changes. These involve changing properties of existing meta-model elements, like the abstractness of a class or the cardinality of a reference. For this reason, we call them *property customizations* and they are also summarized in Table 1.

Regarding classes, we may make abstract a concrete class. This change breaks forward compatibility, and cannot generally be corrected automatically (as e.g., objects of the abstract class may need to be retyped to some subclass). Conversely, this change preserves backward compatibility. Making an abstract class concrete has the converse effect: preserves forward compatibility, while it is an unresolvable backwards.

Regarding inheritance, we may add an inheritance relation between two existing classes. This may break existing models, as instances of the subclass may receive mandatory features owned by the superclass. In general, such changes cannot be automatically corrected. Moreover, this also breaks conformance backwards, as objects of the subclass cease to receive slots typed by features of the superclass. This cannot be corrected automatically, as objects of the subclass may have to be removed from collections (which were compatible with the superclass), and this may break the cardinality of the collection. Removing an inheritance relation has the converse effect.

We can make changes in existing features, regarding its cardinality, type and composition (for references).

With respect to cardinality, we distinguish relaxing or restricting the cardinality interval. In the former case models of the original meta-model still conform to the updated meta-model (forward compatibility) but introduces unresolvable changes for backward compatibility. This is so

because to make the relaxed feature conform back to the original, we may have several choices to remove elements from it. The case of restriction is the converse of relaxation (breaking unresolvable for forward compatibility, and backward compatible). We may have also arbitrary changes to the lower and upper cardinality intervals. This more general change breaks forward and backwards compatibility, and cannot be resolved automatically.

Features (especially references) can also be relaxed or restricted regarding the target type. In the first case, the target of a reference is changed to point to a superclass of the original target. This leads to forward compatibility, but required unresolvable breaking changes for backward compatibility. This is so as it might be impossible to remove a given object from the reference without breaking its cardinality. Reference type restriction changes the target of a reference to point to a subclass, and has the converse effect than reference relaxation. Finally, we may change the target type of a reference by another unrelated type, which is an unresolvable change breaking conformity both forward and backwards.

Finally, a (non-composition) reference may be changed to become a composition reference. This breaks forward compatibility, as the reference in the original model may not have a tree structure. Conversely, it preserves backward compatibility. Making a composition reference non-composition has the converse effect.

Once we have seen the types of possible customizations for a base meta-model, in the next section we propose an approach to specify the space of possible meta-model customizations.

IV. META-MODEL CUSTOMIZATION CONTROL MECHANISM

This section describes the approach we have devised to controlling how a meta-model can be customized. The approach considers all customization types described in Section III (excluding property configurations, left for future work).

As shown in Fig. 6, our method considers two phases. In the first one, the customization rules of the base meta-model are defined. These rules are specified as a model, conforming to a customization meta-model (shown in Fig. 4), which *annotates* the elements of the base meta-model. In the second phase, the base meta-model can be customized according to the established rules. Fig. 6 explicitly depicts the case of meta-model extension. While we support other types of customization, this paper focusses on meta-model extension.

It is common that defining the extension rules and the proper meta-model extension will be performed by different developers. Typically, the definition of the extension rules will be performed by the base meta-model designer (in general by an extensibility designer), while reusing such base meta-model implies its extension according to the specified rules. Both activities should be supported by tools, which need to provide guidance to the developer to extend the meta-model, as well as confidence that the extension made

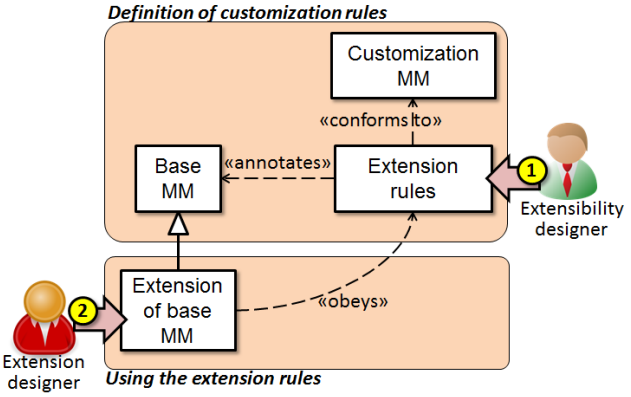

FIGURE 6. Definition and use of customization rules.

TABLE 2. Supported customization rules.

Rule	Applies to	Description	Customization Type
Extend	Class	The tagged class is considered extensible: it is possible to add subclasses	Refinement
Delete	Class, Feature	Permits tagging classes and features as optional	Restriction, Contraction, Restrictive customization
Update	Class, Enumeration	Tags a class or enumeration as "*open*". For classes, they can be added new attributes and references. In case of enumerations, they can be added new literals	Additive customization (Classes), Refinement (Enum)
New	Meta-model	Tags a meta-model as "*open*": it can be added new classes and/or enumeration types	Expansion
Redefine	Reference	Governs how/if references can be redefined	Refinement

obeys the extension rules. Section V gives an overview of the developed tool.

Our approach supports 5 types of customization rules, summarized in Table 2, which cover meta-model extension and customization. Supporting property customizations is left for future work.

Extend rules identify the classes of the base meta-model that can be extended with subclasses. They lead to meta-model refinements (see Fig. 5(a)).

Delete rules tag optional classes or features (attributes and references). For the case of classes, they lead to meta-model restrictions (in the case of optional subclasses) or contractions (for classes with no supertypes).

Update rules can be applied to classes or enumeration types. In the first case, the class is considered open, and new features can be added, leading to (breaking or conservative) additive customizations. In the second, the enumeration type can be added new literals leading to a refinement. Please note

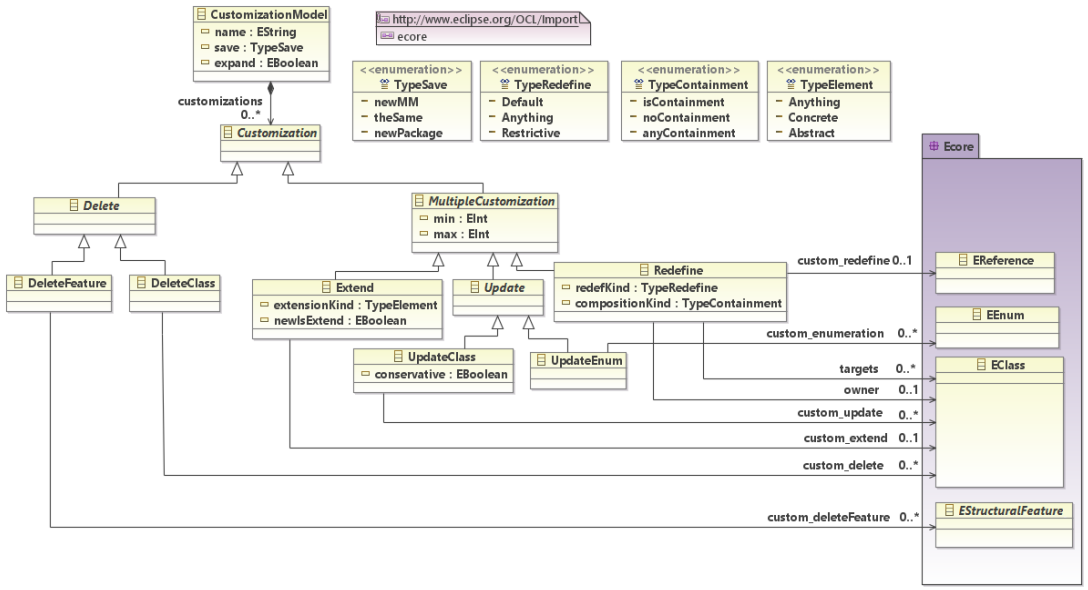

FIGURE 7. Customization meta-model.

that we consider adding new enumeration literals a refinement, as it is a frequent idiom for subclassing a base class.

The *New* rule is applied to the meta-model itself, and indicates that it can be expanded with new classes and/or enumeration types, leading to a meta-model expansion.

Finally, *Redefine* rules are applicable to references of the base meta-model, and govern how they can be redefined in extensions. Redefine rules lead to refinements, as they should be compatible with a suitable *Extend* rule of the owner class or a superclass.

In the scenario of extension of meta-models that we handle, only the *Extend*, *Redefine*, *Enumeration Update*, and *New* rules are relevant.

The customization rules have been realized in a customization meta-model, shown in Fig. 7. The root class of the meta-model is *CustomizationModel*, which holds the name of the extension rule set (*name* attribute), and the policy for storing the customized meta-model (*save* attribute). For the latter, we consider three options (*TypeSave* enum): 1) *newMM:* creates a new meta-model containing both the base meta-model and the extension, 2) *theSame*: modifications are made on the original meta-model, and 3) *newPackage:* the extensions are stored in a separate package referencing the base one. It is possible to indicate that the meta-model is open, so that new independent classes and enumerations can be added (attribute *expand*).

A *CustomizationModel* holds a collection of customization rules. These are instances of some concrete subclass of *Customization*, and are organized in a hierarchy, where the bottom classes (*DeleteFeature*, *DeleteClass*, etc) have a reference to the meta-model element they annotate. The meta-model elements are placed in the package ''Ecore'', because we use the EMF as modeling platform, but this idea is applicable to other meta-modeling platforms as well.

Delete rules (subclasses of *Delete*) tag optional classes and features. Extension, Update and Redefinition rules can all be attached a cardinality interval (min..max), and hence inherit from the abstract class *MultipleCustomization*.

Extension rules allow extending (creating subclasses of) the class of the base meta-model selected by the *custom_extend* reference. The *extensionKind* attribute declares whether the possible subclasses must be abstract, concrete, or left to the judgement of the engineer extending the meta-model. The *newIsExtend* attribute indicates whether the created subclass can be subclassified in its turn. The rule permits specifying the number of subclasses allowed, using the *min..max* interval (where -1 for *max* indicates unlimited). This way, 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, and pointing to the reference to be redefined using *custom_redefine*. 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). Using the *redefKind* attribute, we control the cardinality that can be assigned to each redefinition, considering 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 the *ref* interval. For example, if the cardinality of *ref* is 0..2, then the redefinitions can declare the intervals: 0..1, 0..2, 1..2, 1..1 and 2..2, and 3) *Anything*: the cardinality of the redefinitions can be any.

References may connect two inheritance hierarchies: those of the source and the target classes of the reference. By default, a *Redefine* rule describes how (and if) a reference is redefined when the owner class of the reference is subclassed. Hence, the redefinition rule applies when the owner class or a subclass is extended, and the target class of the redefined reference should be compatible (a subclass) with the target class of the original reference. However, we might want to be more precise in stating both allowed target classes, and when the redefinition should take place.

This way, we can set the *Redefine* rule to apply only when a certain subclass *A* of the reference owner is extended. To describe this possibility, the reference *Redefine.owner* would point to the class *A*. Similarly, we can define the allowed targets of the redefined reference. This is done by reference *Redefine.targets*, which should point to subclasses of the target class of the original reference.

When a reference *ref* is redefined by a series of references ref_1,.., ref_n, reference *ref* is seen as a derived reference, resul-ting from the union of ref_1,.., ref_n. In terms of UML, $\text{ref}_1 \ldots \text{ref}_n$ would have a subsets relation with *ref*.

Example: Fig. 8 shows an example of how redefinition rules work. Part (a) of the figure shows a base meta-model with some extension rules. In particular, they state that class *A* can be extended optionally at most once, while *B* and *C* should be extended mandatorily. In the three cases, the subclasses can be abstract or concrete. Reference *elems* has a redefinition rule, stating that it should be redefined one or more times, whenever *B* (*owner*) is extended. Valid targets of the redefinitions are subclasses of *C*. In addition, redefinitions should be containment (*cont*) and should have * cardinality (*default*).

Fig. 8 (b) shows a valid extension, because *cs* redefines *elems*, and is owned by *B*′, which extends *B*. Moreover, the target of *cs* is correct, as *C*′ is a subclass of *C*. Instead Figs. 8(c) and (d) show invalid extensions. In Fig. 8(c), *bs* is an incorrect redefinition of *elems*, because the target of *bs* is not a subclass of *C*. In Fig. 8(d), *cs* is an incorrect redefinition of *elems*, because the owner of *cs* is not a subclass of *B* (and also because B' does no redefine elems). We will see realistic, practical uses of controlling the owner and target of redefinitions in Section VI.B.

Finally, regarding update rules, there are two kinds: for classes (*UpdateClass*) and for enumeration types (*UpdateEnum*). For classes, they allow adding a number of features in the range (min…max). If the *conservative* attribute is set to true, then only optional features can be added. This makes the rule to be configured for a conservative additive customization, cf. Fig. 5(e). Please note that updating a class means creating new features, and hence there is no need to specify additional *Redefine* rules for owned features. In the case of enumerations, tagging them with *UpdateEnum* means that new literals can be added in the range (min..max).

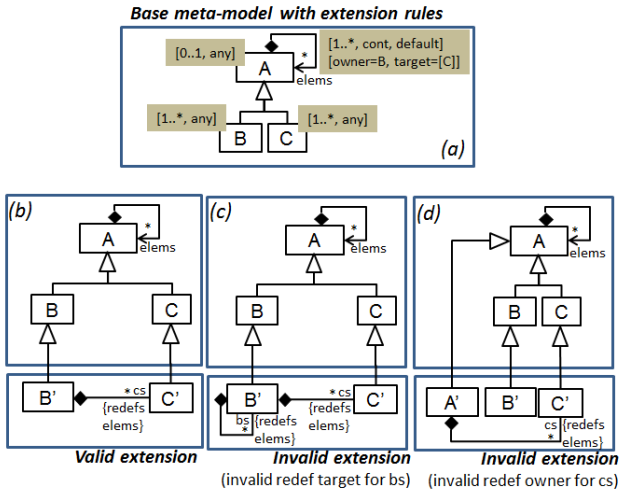

FIGURE 8. Redefinition examples. (a) Base meta-model with extension and redefinition rules. (b) Valid extension. (c, d) Invalid extensions.

The meta-model contains some well-formedness constraints, ensuring a correct placement of extension rules. These include checking that if a reference is tagged as *Redefine*, then the owning class or a subclass has to be tagged as *Extend*, and (if not empty) the *Redefine.owner* reference points to a subclass of the owner of the reference (and similarly for *Redefine.targets*). We also forbid mixing customization intentions, as they may result in possible conflicts. For example, a class tagged as *Delete*, cannot be tagged as *Update* or *Extend*. This is so as, if the class is actually deleted, then it cannot be updated or extended. Similarly, a reference tagged as *Redefine* cannot be tagged as *Delete*. This is so as, if it is redefined, then it would be a conflict to delete it; while if its deleted first it could not be redefined.

Finally, please note that queries (e.g. in OCL) can be defined to identify the kind of customization defined, according to the classification in Table 1. This permits the extensibility designer to control the forward/backward compatibility of models with respect to possible meta-model customizations.

Example: Fig. 9 shows an example illustrating some customization rules in the area of process modeling.

The upper part of Fig. 9 (label a) presents an excerpt of the base meta-model, containing the concepts for an abstract, neutral process modeling language. The lower part (label b) shows its domain-specific extension for software process modeling. Classes *ActivityKind* and *ResourceKind* have attached rules that force their mandatory extension with an abstract class. Hence, these rules force the creation of domain-specific hierarchies for *Activities* and *Resources*. Reference *ActivityKind.needs* must be redefined one or more times with any cardinality, and redefinitions cannot be containments. In the extension, it has been redefined three times in the class *SoftwareEngineeringActivity*, modeling that software engineering tasks: 1) have zero or more inputs which are artefacts; 2) have one or more outputs that are also artefacts;

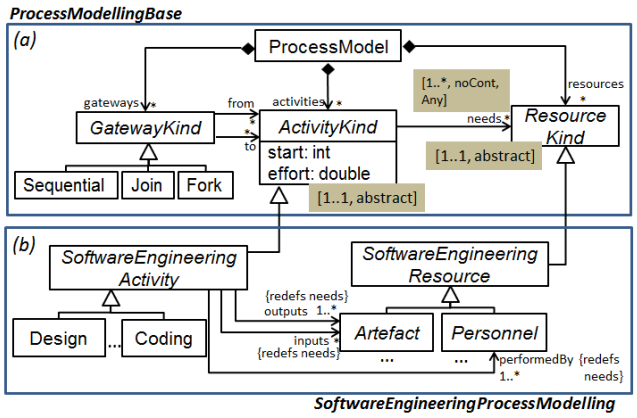

FIGURE 9. Extending a base meta-model to obtain a domain-specific process modeling language.

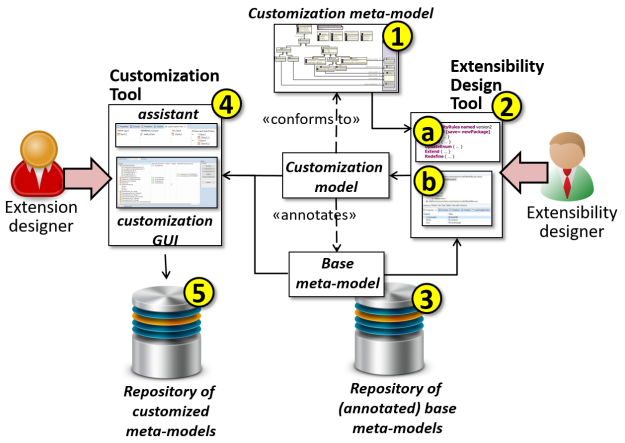

FIGURE 10. Customization architecture.

and 3) are performed by one or more people. The target of each one of these references (*Artefact*, *Personnel*) is compatible with the target of reference *needs (ResourceKind)*. This is the default semantics of redefinition, when no specific target is specified in the *Redefine* rule. This redefinition makes reference *needs* be seen as a derived reference, resulting from the union of references *inputs*, *outputs* and *performedBy*.

Hence, the example has shown that it is possible to design a base meta-model, and control: (a) that only types of activity and resources can be refined, (b) extensions should be organized using an abstract class (like *SoftwareEngineernigActivity* and *ResourceKindActivity*), (c) gateways cannot be refined (e.g., because we assume a fixed semantics for them) and (d) reference *needs* should be redefined in subclasses of *ActivityKind* and *ResourceKind* (mandatorily, to ensure that any domain-specific type of activity requires some domain-specific type of resource). Without a mechanism to define these extensibility rules, desirable extension patterns should be described in natural language, and the extension designer would have the burden to correctly interpret the natural language description, and to check that the designed extension is defined as expected.

An alternative to the definition of a meta-model to express meta-model customizations would be the definition of OCL expressions specifying how a particular meta-model can be extended. However, this would lead to complex, cumbersome expressions, which the extension designer would need to concoct for every new extension specification.

For example, using the previous example, an OCL expression indicating that class *ActivityKind* should be mandatorily extended with an abstract class, could be given as:

```
EClass.allInstances()->one( c |
  c.abstract and
  c.eSuperTypes->exists( c2 |
    c2.name = 'ActivityKind' ))
```

The idea is that, such expression should evaluate to *true* on a valid concrete extension (like *SoftwareEngineering ProcessModelling* in Fig. 9). While the expression could be used to check whether an extension is correct, it cannot be used as a guide to create a suitable extension.

V. TOOL SUPPORT

In order to realize our approach, we have designed an architecture made of a pair of complementary tools that work together, as seen in Fig. 10. These tools are collectively called TACO and have been implemented as Eclipse plugins, using the EMF framework as meta-modeling infrastructure. They are available at https://santiagojacome.wordpress.com/.

The customization architecture uses the customization meta-model (label 1, which was detailed in in Fig. 7), which specifies the operations that can be performed on the meta-model to be customized. The customization rules are created using an *Extensibility Design Tool* (label 2), which permits using either a textual syntax (label a) or a tree-based editor (label b) to create the rules. In both cases, the rules are created as a model (instance of the customization meta-model) which annotates the base meta-model. The base meta-model and the customization rules are stored in a repository (label 3).

Once the extension rules are defined, the base meta-model can be extended according to them. For this purpose, we have created a tool (called *Customization Tool* in the figure, with label 4), which allows executing the operations of the extension model. For this task, the developer is guided by a customization assistant. The resulting customized meta-model can then be stored in a repository (label 5).

Next, we give an overview of the Extensibility Design Tool (Section V.A), and the Customization Tool (Section V.B).

A. EXTENSIBILITY DESIGN TOOL

The customization rules can be specified using a tree-based editor (Fig. 11) or using a Domain-Specific textual language we have created with Xtext (Fig. 12).

Fig. 11 shows an example configuration model with sample values, where:

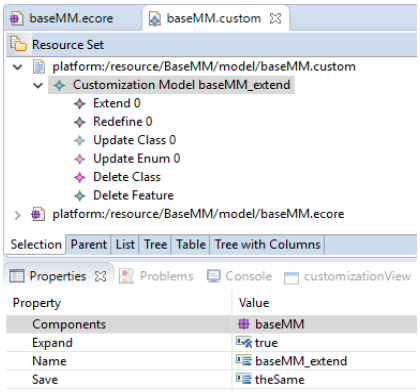

FIGURE 11. Customization model created with the tree-based editor.

```
ExtensibilityRules named baseMM_extend for baseMM [expandable save=theSame]
    ExtClass { ... }
    RedReference { ... }
    UpdClass { ... }
    UpdEnum { ... }
    DelClass { ... }
    DelFeature { ... }
```

FIGURE 12. Customization model created with our textual DSL.

- *Components=baseMM (*.ecore),* is the name of the meta-model to be extended.
- *Expand=true,* controls if new arbitrary classes can be added.
- *Name=baseMM_extend,* is the name of the extension rule set.
- *Save=theSame,* is the storage option of the extended meta-model.
- *Extend, Redefine, UpdateClass, UpdateEnum, DeleteClass* and *DeleteFeature*, is the list of the created customization rules.

If the model specification is made through the extensibility DSL (Fig. 12), initial configuration values of the model must be assigned as in the previous case, as well as the definition of the rules using the language syntax.

B. CUSTOMIZATION TOOL

Once the extensibility rules are specified, they can be used through the customization tool. Its main interface (Fig. 13) is organized in two sections. The left section shows all the elements of the base meta-model (classes, attributes, references and enumerations), which can be modified according to the extension rules defined in the extension model and shown in the columns to the right, next to each element of the base meta-model. The right section of the interface (panel labelled "*OPTIONS*") offers the available extension 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. Additional dialog windows (Fig. 14) are available for the proper execution of each defined rule.

In order to guide the execution of the extension rules, the tool includes a customization assistant as a View.

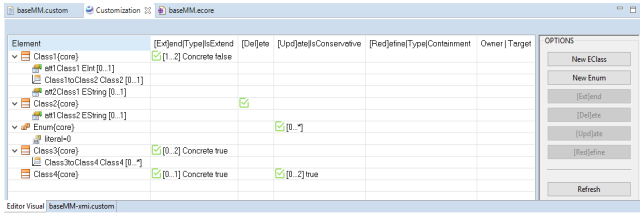

FIGURE 13. Main graphical user interface (GUI) of the customization tool.

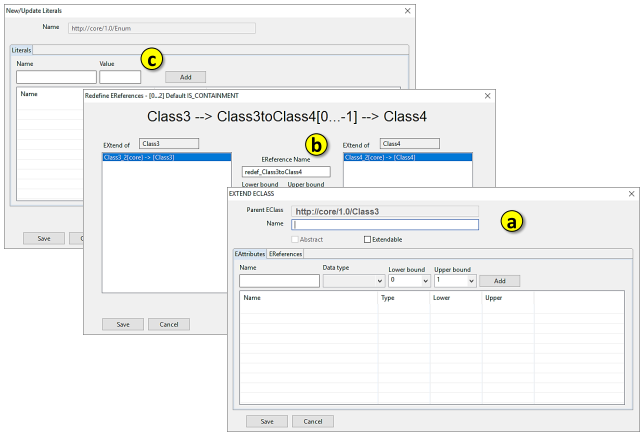

FIGURE 14. Additional dialog windows of the customization tool. (a) Extending a class, (b) Redefining a reference, (c) Adding new literals to an extensible enumeration type.

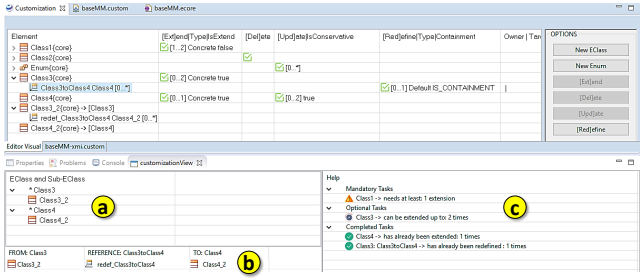

FIGURE 15. Overview of the customization assistant.

The assistant provides help for the process of class extension and for the process of redefining references (Fig. 15).

The assistant appears as a view divided into three sections at the bottom of the main interface. In the process of class extension, selecting a class from the *Element* column of the main GUI shows all its subclasses, giving an overall idea of the inheritance hierarchy (section a in Fig. 15). In the process of redefining references, selecting a reference from the *Element* column of the main GUI shows the source and destination class of the reference, and the name of the reference being redefined. All subclasses of the source and target classes of the reference to be redefined are also displayed (section b of the Figure).

Finally, section c of the assistant guides in the creation of the extension. This way, it contains three parts: mandatory tasks, optional tasks and completed tasks.

Mandatory tasks are customizations required by some of the rules. For example, an *Extend* rule with a cardinality of [1..2] is a mandatory task. Fig. 15 shows that *Class1* needs to be extended at least once.

Optional tasks are tasks that either have a cardinality with 0 as lower bound or are mandatory tasks that have already been performed. For example, in the Figure, if we extend *Class1* once, then extending it another time becomes an optional task. Extending *Class3* (which has cardinality [0..3]) is an optional task from the beginning. Finally, completed tasks are customization actions performed by the developer.

VI. EVALUATION

Next, we evaluate the approach under two perspectives. First, we analyse the applicability of our approach, by evaluating to what extent meta-model extensibility is a common need, understand the rationale for extensibility, and check if our approach can be used to express extensibility over meta-models built by third parties. Second, we present a relevant case study, taken from the DD OMG standard. In this case study, we define different extensibility rules, which can be used to define different types of allowed extensions.

A. APPLICABILITY OF THE APPROACH

This section analyses to what extent our approach is useful to express extensibility rules found in practice, and to understand practical scenarios that benefit from extensibility control.

Table 3 shows some meta-models from different sources, and how we described their extensibility rules using our approach. We have investigated two different repositories: the ATL meta-model zoo,[2] and publicly available OMG standards. Additionally, we have also analyzed meta-models found in conferences and journals in the modeling area.

It can be observed that extensibility needs are found in the ATL zoo, the OMG standard and papers found in the literature [4], [28]–[32]. In these works, we observe three kinds of needs. In the first one, one would like to have library-like mechanisms at the meta-model level, e.g., to define extensible catalogs of activity kinds (like in Intalio BPMN), diverse policy types (like in MARTE), document properties (like in SACM [33]) or types of services (like in PetStore). In this scenario, several classes or enumeration types could be tagged as ''open'' using our approach.

In the second scenario, a base language is provided, which needs to be extended with domain-specific concepts. In some cases, meta-models for external technologies are expected to be plugged-in extending some designated class(es). Typical examples are meta-models in the area of reverse engineering (like KDM or ASTM) or enterprise application integration (like EAI) where base meta-models are expected to be extended with meta-models of programming languages. In some cases, the extension developer needs to follow complex patterns for the extension (a typical case is KDM [9]), which are described in natural language only.

Finally, in the third scenario, there is a need to dynamically instantiate types and their instances. These cases are either modelled using the type-object pattern [25], [27] or promotion transformations [26]. Using our approach permits using subclassing, ensuring a controlled extension of a base meta-model. An alternative to using subclassing would be the use of multi-level modeling, as discussed in Section II.B and [4].

As an example of the benefits that our approach would bring in the second scenario, Fig. 16 shows an excerpt of the EMF-DEVS meta-model, and a small fragment of an extension [25]. This meta-model pertains to the area of component-based modeling and simulation. A base model (EMF-DEVS) is defined, and the designer is expected to define particular types of components (with input and output ports) and domain-specific data types for the exchanged messages

[2]http://web.emn.fr/x-info/atlanmod/index.php?title=Main_Page

TABLE 3. Extensible meta-models (CL=Class Extension, EN=Enumeration Extension, RE=Reference Redefinition).

Meta-model	Source	Extensibility rules used
EMF-DEVS	Sarjoughian [22, 23]	**CL**: eAtomic, eIOData, eInPort, eOutPort, eCoupled, eEOCoupling, eICoupling, eEICoupling. **RE**: outPorts, inPorts, values.
DSLs4BPM	Heitkötter [24]	**CL:**ProcessBuildingBlock, OrganizationalElement.
Security Policies	Mouelhi [25]	**CL:** PolicyType, RuleType, ElementType, HierarchicalElement. **RE:** ruleTypes, elementTypes, parameters.
TML	Drivalos [26]	**CL:** Trace, TraceLink, Context. **RE:** context, traceLinks
CloudML	CloudML [27]	**CL**: Artafact, ServerPort, Node. **RE**: provider.
Case representation	ATL zoo	**EN**: ParameterKind.
Intalio BPMN	ATL zoo	**EN**: ActivityKind.
LQN 1.0	ATL zoo	**EN**: LinkType, PhaseType.
XPDL	ATL zoo	**EN**: GatewayTypeKind.
SWRC	ATL zoo	**CL**: Publication, Event, Person, Topic, Organization, Project.
PetStore	ATL zoo	**CL**: Services.
KDM	OMG	**CL**: KDMModel, KDMEntity, KDMRelationship. **RE**: ownedElement, ownedRelationship, outbound, inbound.
ASTM	OMG	**CL:** OtherSyntaxObject, Definition, DataType, Statement, Expression.
DD	OMG	**CL:** Edge, Shape, DiagramElement, Diagram, Style. **RE:** localStyle, sharedStyle, source, target. **EN:** ColorKind.
EAI	OMG	**CL**: TDLangClassifier, TDLangComposedType, TDLangElement.
BQS1.0	OMG	**CL:** BibliographicReference, Provider.
SACM 1.0	OMG	**CL:** DocumentProperty, EvidenceEvent, Provenance, EvidenceAttribute.
MARTE	OMG	**EN:** SynchronisationKind, PoolMgtPolicyKind, PoolMgtPolicy, SchedPolicyKind, AccessPolicyKind, ProtectProtocolKind, PLD_Class, QueuePolicyKind, PLD_Technology, MutualExclusionResourceKind, ConcurrentAccessProtocolKind, NotificationResourceKind, ROM_Type, CacheType, LaxityKind, MessageResourceKind, ComponentState, PortType, ConditionType, OptimalityCriterionKind.

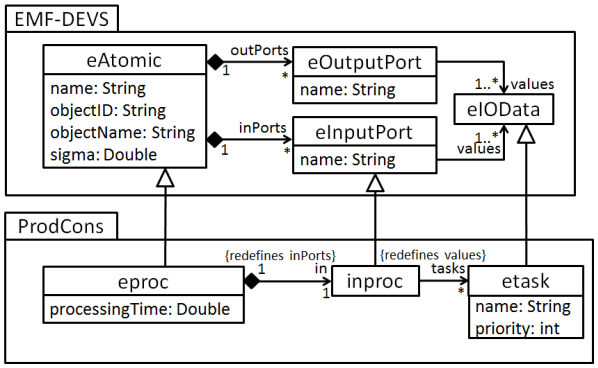

FIGURE 16. Excerpt of EMF-DEVS meta-model, and a typical extension (adapted from [25]).

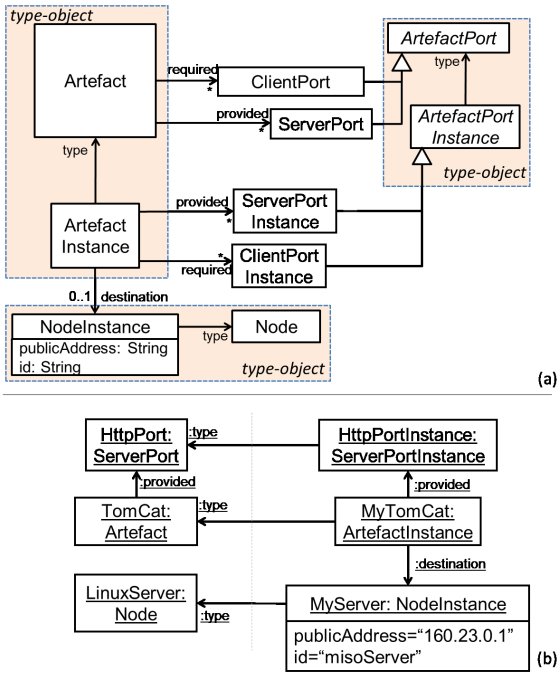

FIGURE 17. Excerpt of the CloudML meta-model (a), and an instance (b).

by subclassing from *eAtomic*, *eOutputPort*, *eInputPort* and *eIOData*.

Hence, the designer is not expected to modify existing classes in EMF-DEVs, while subclassing *eAtomic* at least once, and redefining the association between *eAtomic* and the port whenever such port is subclassified. Without our approach, such rules would be expressed with natural language, while our tool allows describing them, guiding the developer in the extension and guaranteeing a correct extension according to the rules.

As an example of the third scenario, Fig. 17 shows a small part of the CloudML meta-model (part a), and a small example instance (part b). This meta-model has been designed using the type-object pattern (the instances of such pattern have been signalled in the Figure), as there is the need to leave users the definition of new types (e.g., new kinds of

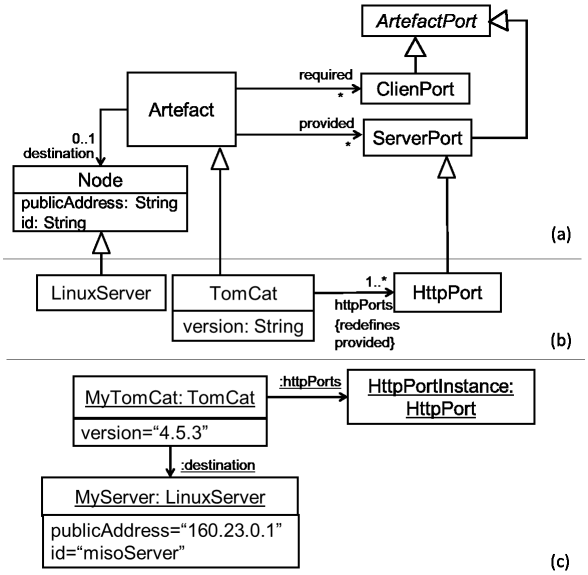

FIGURE 18. Excerpt of the CloudML meta-model rearchitected using our extension-based approach (a), and an instance (b, c).

Artefact like *TomCat*) and instances of these (e.g., instances of *ArtefactInstance* like *MyTomCat*). To obtain extensibility, the decision is to model types (like *TomCat*) as objects. This has the drawback that, should we need these types to define new properties (as it is indeed the case in the meta-model), these attributes should be emulated with an additional class in the meta-model.

Instead, we can rearchitect CloudML using our extensibility approach, as shown in Fig. 18 (a). By using extensibility rules, users of CloudML can safely extend the base meta-model in a modular way, with no risk of performing incorrect extensions. As a clear advantage of our approach, the base CloudML meta-model is reduced in size (5 classes and 3 associations vs. 10 classes and 8 associations), as each occurrence of the type-object pattern is represented with one class only. Moreover, by modeling types at the meta-model level, we can naturally add properties to such types (e.g., version for *TomCat*), and specify cardinalities for redefined associations (e.g. for *httpPorts*). Hence, our approach permits users to perform a kind of domain-specific meta-modeling [34].

B. CASE STUDY: DEFINING EXTENSION RULES FOR DI

In order to further assess the feasibility of the proposed approach, in realistic scenarios, we have defined extensibility rules on several meta-models (as described in the previous subsection), including standard meta-models such as DD and KDM. The extension rules are available at the tool's web site: https://santiagojacome.wordpress.com/.

This section describes in some more detail the extension process for DD v1.1. The objective of this study is to define the rules of extension of the DI meta-model (Fig. 19) to create an extension of it for UML (similar to the one proposed in [15]) , adhering to the architecture of Fig. 1.

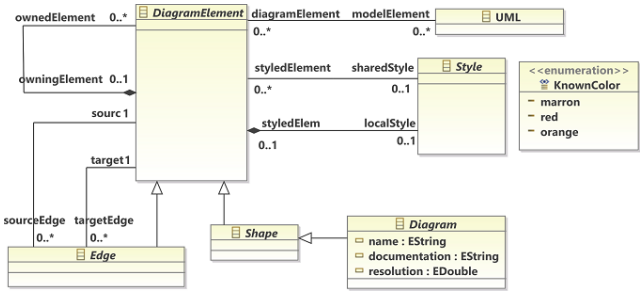

FIGURE 19. Diagram Interchange meta-model.

TABLE 4. Extension rules for DI.

Extension rules		Value	Application examples
Extend: 1 (Diagram)	ExtensionKind Min, Max NewIsExtend	Concrete 1, 1 true	Class *UMLDiagram* created.
Extend: 2 (DiagramElement)	ExtensionKind Min, Max NewIsExtend	Abstract 0, * true	Class *UMLDiagramElement* created.
Extend: 3 (Edge)	ExtensionKind Min, Max NewIsExtend	Anything 1, * false	Class *UMLEdge* created. *UMLEdge* inherits from *UMLDiagramElement*.
Extend: 4 (Shape)	ExtensionKind Min, Max NewIsExtend	Anything 1, * false	Classes *UMLCompartment, UMLShape* (with attr *showClassifierShape*) and *UMLLabel* (with attr *showQualified*) created. *UMLShape* inherits from *UMLDiagramElement*. *UMLLabel* (with attr showQualified) created.
Extend: 5 (Style)	ExtensionKind Min, Max NewIsExtend	Concrete 1, 1 true	Class *UMLStyle* (with attr *fontName* and *fontSize*) created.
Redefine: 1 (localStyle)	RedefKind CompositionKind Min, Max owner targets	Default Containment 1, 1 -- --	Reference *umlLocalStyle* between *UMLDiagramElement* and *UMLStyle* created.
Redefine: 2 (sharedStyle)	RedefKind CompositionKind Min, Max owner targets	Default noContainment 1, 2 -- --	Reference *umlSharedStyle* between *UMLDiagramElement* and *UMLStyle*.
Redefine:3 (ownedElement)	RedefKind CompositionKind Min, Max owner targets	Default Containment 1, * Diagram --	Reference *umlOwnedElement* betwen *UMLDiagram* and *UMLDiagramElement*.
Redefine:4 (ownedElement)	RedefKind CompositionKind Min, Max owner targets	Default Containment 0, * -- --	Not used.
Redefine:5, 6 (source, target)	RedefKind CompositionKind Min, Max owner targets	Default noContainment 1, 1 -- --	References *sourceShape* and *targetShape* between *UMLEdge* and *UMLShape*.
UpdateEnum:1 (KnownColor)	Min, Max	0, 10	Add custom colors to *KnownColor*.

According to the standard, extensions to this package must be made using a uniform pattern. The problem is that the DD 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).

Table 4 shows the extension rules, extracted from the natural language indications of the standard. The table has four columns, indicating the type of rule and the base meta-model

```
ExtensibilityRules named DI_extend for DI [expandable save=newPackage]
ExtClass { DI.core.Diagram [1..1] concrete extensible}
ExtClass { DI.core.DiagramElement [0..*] abstract extensible}
ExtClass { DI.core.Edge [1..*] anything }
ExtClass { DI.core.Shape [1..*] anything }
ExtClass { DI.core.Style [1..1] concrete extensible }
RedReference { DI.core.DiagramElement.localStyle [1..1] isContain }
RedReference { DI.core.DiagramElement.sharedStyle [1..2] noContain }
RedReference {DI.core.DiagramElement.ownedElement [1..*] isContain owner
DI.core.Diagram}
RedReference { DI.core.Edge.sourc [1..1] noContain }
RedReference {DI.core.Edge.target [1..1] noContain }
UpdEnum { DI.core.KnownColor [0..10] }
```

FIGURE 20. Extension rules using the extensibility DSL.

element they apply to, the value of each field of the rule, and an example of use to create an extension for the UML. Fig. 20 shows the same rules in textual format.

Altogether, we consider 5 extension rules, 6 redefinition rules, and 1 update enumeration rule.

The first rule specifies that the class *Diagram* should be extended mandatorily by a concrete class. This is necessary to define a specific type of diagram for the considered modeling language. Classes *Shape* and *Edge* should also be refined mandatorily (by concrete classes) to define domain-specific shapes and edges. *DiagramElement* can optionally be extended, the rationale being that it may be useful to define a base class for the different domain-specific elements. Hence, we define an optional extension by abstract classes. Finally, *Style* should also be extended, to define domain-specific styles.

The most interesting redefinition rules are for *ownedElement* (redefinition rules 3 and 4). Redefinition rule 3 specifies that redefining *ownedElement* is mandatory for extensions of *Diagram*. This is necessary to enable inclusion of shapes and edges in the diagram. Redefinition rule 4 states that such redefinition is optional for extensions of other subclasses of *Diagram* (like *Node* and *Shape*), as that would only be needed to model shape containment. Please note that while both redefinition rules apply to *ownedElement*, rule 3 takes precedence when extending the *Diagram* class (as it applies specifically to subclasses of *Diagram*).

Finally the *UpdateEnum* rule specifies that enumeration *KnownColor* can be updated with new custom colors.

Once the extensibility rules are specified, they can be used through the main GUI (Fig. 21). As noted, it is organized into two sections, the first column of the left section shows all the elements of the base meta-model, which can be extended 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 newly added elements are shown in colored rows at the bottom of the table.

Fig. 22 shows the resulting extension, built with the extension GUI (classes and references in red). This extension is for the UML language, and is an excerpt of that defined in the standard itself, and in [15]. As it can be observed, we have defined an extension *UMLDiagram*, which redefined *ownedElement* to store *UMLDiagramElements*. *UMLDiagramElement* is the base class for UML shapes and edges.

Altogether the use of our approach ensures that any extension the DI meta-model is defined as expected by the

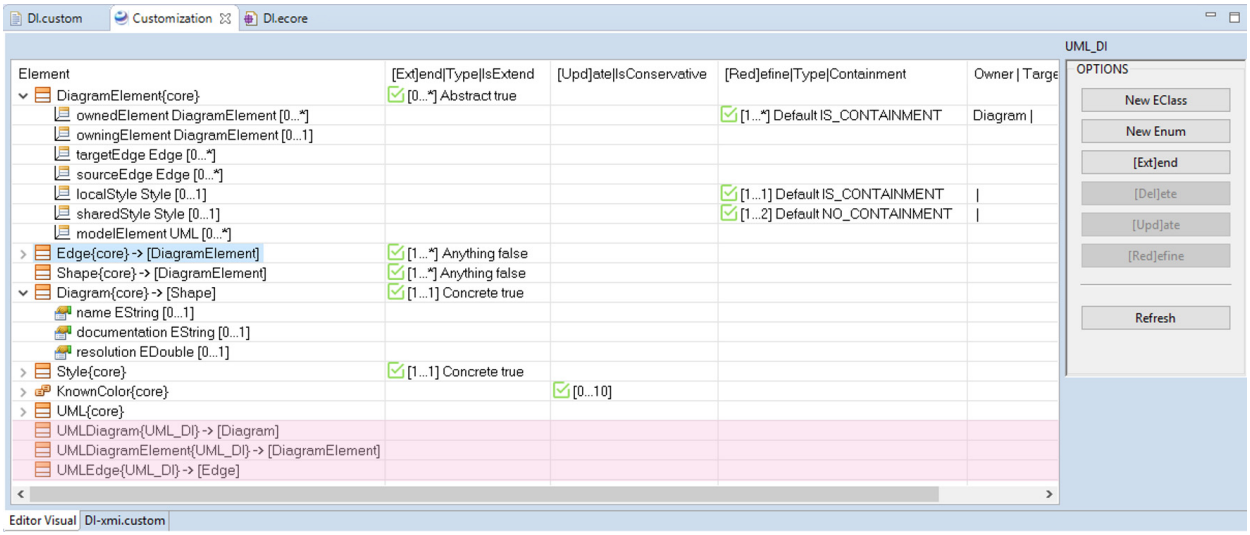

FIGURE 21. Extending the DI meta-model for UML through the customization tool.

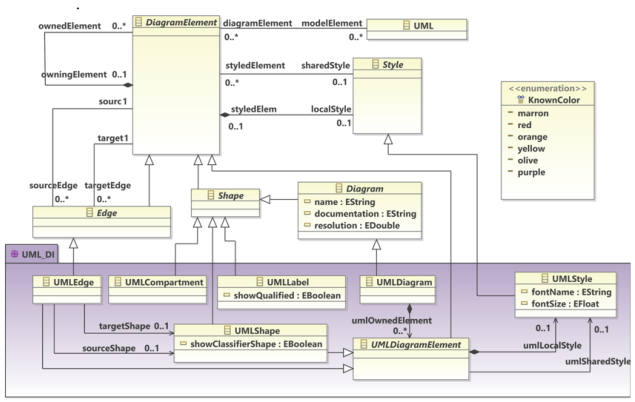

FIGURE 22. Excerpt of the extended DI meta-model for UML.

DD designers. In addition, our approach permits defining different types of extensibility. We call them extensibility *profiles*. For example, in the case of DI, we can define extensibility rules that result in flat graph-like diagrams [35]. For this profile, we would like to forbid nesting of edges and shapes (which are allowed by the redefinition rule 4 in Table 4), and forbid edges connecting edges (allowed by the redefinition rules 5 and 6 in Table 4). This way, the extension designer can chose the more appropriate extension profile, while those profiles still leave room for variability in the extensions.

Table 5 shows the modified redefinition rules for this extensibility profile, which in addition contains all rules defined in Table 4, except the redefinition rule 4. Table 5 contains two rules, applying to references *source* and *target*. Both rules are shown together for brevity, and demand that redefinition of both such references should point to extensions of Shape (hence avoiding pointing to extensions of *Edge*).

As an example, Fig. 23 shows an example extension made using the extensibility profile.

TABLE 5. "Simple Graph Diagram" extensibility profile for DI.

Extension rules		Value	Application examples
Redefine:5, 6 (source, target)	RedefKind	Default	Redefined references *sourceShape* and *targetShape* between *UMLEdge* and *UMLShape*.
	CompositionKind	noContainment	
	Min, Max	1, 1	
	owner	--	
	targets	Shape	

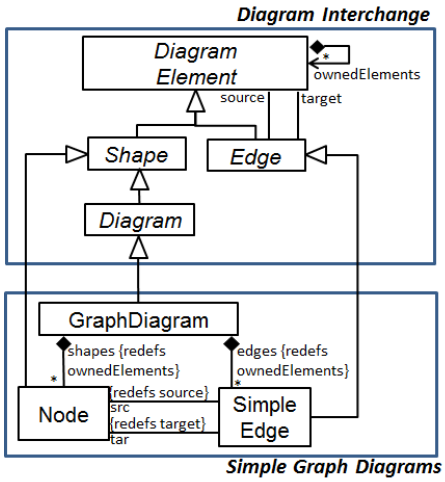

FIGURE 23. Using the DI extensibility profile for graph-like diagrams.

Nested shapes and edges are forbidden by allowing refinements of reference *ownedElements* for subclasses of *Diagram* only (redefinition rule 4 was removed). This way, no subclass of *Node* or *SimpleEdge* can redefine reference *ownedElement*. Moreover, domain-specific types of edges cannot connect other edges due to the new redefinition rules 5 and 6 in the profile. Please note also that these rules require a multiplicity of 1..1 for the redefined references (default option), hence disabling the possibility of creating hyperedges (connecting multiple sources to multiple targets).

Altogether we can see that our mechanism to define extensibility rules can guide the extension designer in constructing correct extensions, as expected by the DD standard. Moreover, our approach would enable the creation of different extensibility profiles, so that different extension styles can be chosen.

VII. 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 [12] 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 meta-model development time. The work defines a set of extension operators that support the creation and modification of classes (modify operators, add, modify, and filter properties), as well as to establish specialization and generalization relations between classes. 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 [36] it is pointed out that UML profiles provide a means for adapting existing meta-models to specific platforms. The UML profiles constitute a light extension mechanism where it is possible to extend the meta-model without overwriting the original elements [37]. The profiles allow to extend the meta-models through stereotypes, constraints, and tagged values. These three extension mechanisms are not top-level, namely they do not allow modifying existing meta-models, only adding elements and constraints, but respecting the original syntax and semantics [38]. In [39] the philosophy of extending profiles 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.

Braun [40] provides a classification for extension mechanisms and considers the concept of *hook* to leave open parts of a program in order to define and specify them later in classes, interfaces or methods that must be concretized for injection of code [41], while other parts of the software remain fixed, called *frozen points* in [42]. Braun and Esswein [10] explore several MOF meta-model extension mechanisms, based on an analogy with the principles of extension in the field of software engineering, such as: *hook*, *aspects*, *plug-ins*, and *add-ons*. An approach like ours would allow to specify allowed extensions.

In [43] two types of meta-model extension mechanisms are proposed and defined as a *controlled mechanism* and *uncontrolled mechanism*. In the uncontrolled extension mechanism, elements can be added to the original meta-model arbitrarily and associations can be defined between two constructs of the meta-model. Whereas in the controlled mechanism any new construction of the meta-model must be derived from the original meta-model and only when a new association is a specialization of an existing one in the original meta-model could be added to the meta-model. Our approach could be complementary, allowing the definition of precise rules for both mechanisms.

In [44] four levels of the extension mechanism of the UML model are defined and discusses the ability to read, expressiveness, reach of use and support of tools based on precise definitions of levels of extension. Different levels mean different expression abilities and application limits. When modellers extend the UML meta-model according to their actual modeling requirements, they can select an appropriate extension level to make the extension more reasonable and operable.

Techniques borrowed from software product lines have been proposed to express variants of DSLs [45]. However, such techniques express a *closed*, predefined set of DSL variants, while our extension rules permit describing a possibly infinite set of extensions, all of which should obey the extensibility rules. Hence, we argue that our approach constitutes an *open,* but controlled mechanism for extension.

In some cases, extensibility is required due to an intrinsic need to use multiple meta-levels. In [4], several patterns were identified, indicating that using multiple meta-levels would be advantageous. In particular, the type-object pattern [4] described using the ''*static types*'' and '' *enumeration*'' approaches would indicate a need for extensibility. In the first case, one or several classes are expected to be subclassified, and some references stemming from it redefined. In the second case, an enumeration type is ''open'', in the sense that it would admit further literals. Both cases can be covered with our approach.

Guerra and one of the authors of this paper proposed a flattening of multi-level models which replaces instantiation by inheritance, for the purpose of analyzing their instantiability [46]. Our extensibility approach replaces instantiation by inheritance, but, because we leave the user to perform such extensions, we propose extensibility rules to facilitate this task and ensure its correctness.

Atkinson *et al.* [3] points to 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 eliminate the weaknesses of the mechanisms outlined above.

Overall, in summary, we can see a large number of works analysing language extension mechanisms or DSLs, but there is a lack of mechanisms to define extension rules for them in an open way (i.e., leaving freedom for customization to the user, but following certain rules), an aspect in which our work is novel and complements these existing works.

VIII. CONCLUSIONS

In this article we have proposed a mechanism, an architecture, and a set of tools supporting the definition of extension rules for meta-models, and making 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. To complement meta-model extension and customization, we will tackle the problem of backward/forward model migration. We are also planning to perform a usability study of the tool. Finally, we will extend the tool to handle other scenarios, including multi-level modeling and adaptation of DSLs.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] S. J. Mellor, T. Clark, and T. Futagami, "Model-driven development: Guest editors' introduction," *IEEE Softw.*, vol. 20, no. 5, pp. 14–18, Sep./Oct. 2003.
[2] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.
[3] C. Atkinson, R. Gerbig, and M. Fritzsche, "A multi-level approach to modeling language extension in the enterprise systems domain," *Inf. Syst.*, vol. 54, pp. 289–307, Dec. 2015.
[4] J. de Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, p. 12, 2014.
[5] (2011). *Knowledge Discovery Meta-Model (KDM), Version 1.3*. [Online]. Available: http://www.omg.org/spec/KDM/1.3/PDF/
[6] (2015). *Diagram Definition (DD)*. [Online]. Available: http://www.omg.org/spec/DD/
[7] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Reading, MA, USA: Addison-Wesley, 2009.
[8] S. Jácome-Guerrero and J. De Lara, "Un enfoque para controlar la extensión de meta-modelos en el ámbito de la Ingeniería dirigida por modelos," CIbSE, Buenos Aires, Argentina, Tech. Rep., 2017, pp. 29–42.
[9] S. P. Jácome-Guerrero and J. De Lara, "Towards a mechanism for controlling meta-model extensibility," ICSOFT, Madrid, Spain, 2017, pp. 382–387.
[10] R. Braun and W. Esswein, "Extending the MOF for the adaptation of hooks, aspects, plug-ins and add-ons," in *Model and Data Engineering*. Springer, 2015, pp. 28–38.
[11] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 493–509, Jul. 1999.
[12] H. Bruneliere *et al.*, "On lightweight metamodel extension to support modeling tools agility," in *Proc. Eur. Conf. Modeling Found. Appl.*, 2015, pp. 62–74.
[13] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Commun. ACM*, vol. 40, no. 10, pp. 32–38, 1997.
[14] *Documents Associated With XML Metadata Interchange (XMI), Version 2.4.1*. Accessed: Jun. 2013. [Online]. Available: http://www.omg.org/spec/XMI/2.4.1/
[15] M. Elaasar and Y. Labiche, "Diagram definition: A case study with the UML class diagram," in *Model Driven Engineering Languages and Systems*. Berlin, Germany: Springer, 2011, pp. 364–378.
[16] C. Bock and M. Elaasar, "Reusing metamodels and notation with diagram definition," in *Proc. Softw. Syst. Modeling*, 2016, pp. 1–25.
[17] J. De Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Proc. Int. Conf. Model. Techn. Tools Comput. Perform. Eval.*, 2010, pp. 1–20.
[18] C. Atkinson and R. Gerbig, "Flexible deep modeling with melanee," in *Proc. Modellierung (Workshops)*, 2016, pp. 117–122.
[19] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara, "Pattern-based development of domain-specific modelling languages," in *Proc. ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS)*, Sep. 2015, pp. 166–175.
[20] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *Proc. 5th Int. Conf. Generat. Programm. Compon. Eng.*, 2006, pp. 211–220.
[21] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Proc. 12th Int. IEEE Enterprise Distrib. Object Comput. Conf. (EDOC)*, Sep. 2008, pp. 222–231.
[22] H. S. Sarjoughsian, A. Alshareef, and Y. Lei, "Behavioral DEVS meta-modeling," in *Proc. Winter Simulation Conf.*, 2015, pp. 2788–2799.
[23] H. S. Sarjoughian and A. M. Markid, "EMF-devs modeling," in *Proc. Symp. Theory Model. Simulation-DEVS Integr. M&S Symp.*, 2012, p. 19.
[24] H. Heitkötter, "A framework for creating domain-specific process modeling languages," in *Proc. ICSOFT*, 2012, pp. 127–136.
[25] T. Mouelhiv, F. Fleurey, and B. Baudry, "A generic metamodel for security policies mutation," in *Proc. IEEE Int. Conf. Softw. Test. Verification Validat. Workshop (ICSTW)*, Apr. 2008, pp. 278–286.
[26] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes, "Engineering a DSL for software traceability," in *Proc. Int. Conf. Softw. Lang. Eng.*, 2008, pp. 151–167.
[27] (2018). *CloudML*. [Online]. Available: http://cloudml.org/
[28] G. Deltombe, O. Le Goaer, and F. Barbier, "Bridging KDM and ASTM for model-driven software modernization," in *Proc. SEKE*, 2012, pp. 517–524.
[29] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. Kelly, "Weaving an assurance case from design: A model-based approach," in *Proc. IEEE 16th Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2015, pp. 110–117.
[30] *Negotiation Facility—NEG, Version 1.0*, Object Manage. Group, Needham, MA, USA, 2002.
[31] *UML Profile for Enterprise Application Integration (EAI), Version 1.0*, Object Manage. Group, Needham, MA, USA, 2004.
[32] *Structured Assurance Case Metamodel (SACM), Version 2.0 Beta*, Object Manage. Group, Needham, MA, USA, 2017.
[33] J. L. de la Vara, G. Génova, J. M. Álvarez-Rodríguez, and J. Llorens, "An analysis of safety evidence management with the structured assurance case metamodel," *Comput. Standards, Interfaces*, vol. 50, pp. 179–198, Feb. 2017.
[34] J. de Lara, E. Guerra, and J. S. Cuadrado, "Model-driven engineering with domain-specific meta-modelling languages," *Softw., Syst. Model.*, vol. 14, no. 1, pp. 429–459, 2015.
[35] P. Bottoni and A. Grau, "A suite of metamodels as a basis for a classification of visual languages," in *Proc. IEEE Symp. Vis. Lang. Hum. Centric Comput.*, Sep. 2004, pp. 83–90.
[36] R. Braun and W. Esswein, "Towards an integrated method for the extension of MOF-based modeling languages," in *Model and Data Engineering*. Springer, 2015, pp. 103–115.
[37] J. E. Pérez-Martínez, "Heavyweight extensions to the UML metamodel to describe the C3 architectural style," *ACM SIGSOFT Softw. Eng. Notes*, vol. 28, no. 3, p. 5, 2003.
[38] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault, "A practical approach to bridging domain specific languages with UML profiles," in *Proc. Best Practices for Model Driven Softw. Develop. (OOPSLA)*, San Diego, CA, USA, 2005, p. 2005.
[39] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "EMF profiles: A lightweight extension approach for EMF models," *J. Object Technol.*, vol. 11, no. 1, pp. 1–29, 2012.
[40] R. Braun, "Behind the scenes of the BPMN extension mechanism principles, problems and options for improvement," in *Proc. 3rd Int. Conf. Model-Driven Eng. Softw. Develop. (MODELSWARD)*, 2015, pp. 1–8.
[41] D. Birsan, "On plug-ins and extensible architectures," *Queue*, vol. 3, no. 2, pp. 40–46, 2005.
[42] U. Kulesza, V. Alves, A. Garcia, C. J. De Lucena, and P. Borba, "Improving extensibility of object-oriented frameworks with aspect-oriented programming," in *Proc. Int. Conf. Softw. Reuse*, 2006, pp. 231–245.

[43] A. Schleicher and B. Westfechtel, "Beyond stereotyping: Metamodeling approaches for the UML," in *Proc. 34th Annu. Hawaii Int. Conf. Syst. Sci.*, 2001, p. 10.
[44] Y. Jiang, W. Shao, L. Zhang, Z. Ma, X. Meng, and H. Ma, "On the classification of UML's meta model extension mechanism," in *Proc. Int. Conf. Unified Modeling Lang.*, 2004, pp. 54–68.
[45] G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, and P.-Y. Schobbens, "Featured model types: Towards systematic reuse in modelling language engineering," in *Proc. IEEE/ACM 8th Int. Workshop Proc. Modeling Softw. Eng. (MiSE)*, May 2016, pp. 1–7.
[46] E. Guerra and J. de Lara, "Automated analysis of integrity constraints in multi-level models," *Data, Knowl. Eng.*, vol. 107, pp. 1–23, Jan. 2017.

SANTIAGO JÁCOME received the Engineering degree in computer systems from the Escuela Politécnica Nacional, Ecuador, and the M.S. degree in research and innovation in formation and communication technologies from the Universidad Autónoma de Madrid, where he is currently pursuing the Ph.D. degree in computer and telecommunication engineering.

He is currently a Professor-Researcher in software engineering with the Universidad de las Fuerzas Armadas ESPE, Ecuador. His research interests are in model-driven engineering specifically in meta-modeling, domain-specific languages, and multi-level modelling. He currently collaborates with the Modeling and Software Engineering Research Group.

JUAN DE LARA received the Ph.D. degree in computer science from the Universidad Autónoma de Madrid. He is currently a Professor with the Computer Science Department, Universidad Autónoma de Madrid, where he leads the Modeling and Software Engineering Research Group. He has been a Post-Doctoral Researcher/Visiting Professor with the MSDL lab, McGill University, the Institute of Theoretical Computer Science, TU Berlin, the Department of Computer Science of the University of Rome "Sapienza," the University of York, U.K., and the University of Toronto, Canada. His research interests are in meta-modeling, multi-level modeling, domain-specific languages, and model transformation. He has published over 170 papers in international journals and conferences in these areas.

Dr. de Lara has been the PC Co-Chair of conferences, such as ICMT, ICGT and FASE. He is currently on the Editorial Board of *Software and Systems Modeling*.